INTERNATIONAL CENTER FOR NUMERICAL METHODS IN ENGINEERING

Implementation of a General Algorithm for Incompressible and Compressible Flows within the Multi-Physics code KRATOS and Preparation of Fluid-Structure Coupling

M. May
R. Rossi
E. Oñate

Publication CIMNE Nº-327, November 2008

Implementation of a General Algorithm for Incompressible and Compressible Flows within the Multi-Physics code KRATOS and Preparation of Fluid-Structure Coupling

M. May
R. Rossi
E. Oñate

Publication CIMNE Nº-327, November 2008

International Center for Numerical Methods in Engineering
Gran Capitán s/n, 08034 Barcelona, Spain

Statutory Declaration

I hereby formally declare that I myself have written the enclosed diploma thesis independently. I did not use any sources or means without declaration in the text; any thoughts from others or literal quotations are clearly marked.
To the best of my knowledge and belief, I confirm that this thesis, either completely or in parts, has not been previously submitted to achieve an academic grading elsewhere.

Munich, 16. May 2008

Markus May, Student-ID #2456241

Acknowledgements

This diploma thesis has been created in cooperation with the International Center for Numerical Methods in Engineering (CIMNE) at the Universitat Politècnica de Catalunya in Barcelona.

First of all, I would like to thank Prof. Eugenio Oñate, director of the CIMNE, for the opportunity of doing my diploma thesis in Barcelona and for the very professional and inspiring working environment at his institute. In this context I would like to express my special gratitude to Dr. Riccardo Rossi for the neverending encouragement, the technical advice and the interest in my work. Furthermore, I want to thank Dr. Enrique Ortega and Dr. Mariano Vazquez for their explanations on the simulation of compressible flows and Dr. Pooyan Dadvand for his support regarding programming issues in Kratos. In addition, Prof. Roberto Flores always offered a helping hand providing his knowledge on aerospace engineering. I also found a very patient advisor in Prof. Ramon Codina who helped me in answering difficult mathematical questions and whose office has always been open to me.

At the Technische Universität München, Prof. Dr.-Ing. Wolfgang A. Wall, head of the Chair of Computational Mechanics, and Dr.-Ing. Volker Gravemeier spontaneously assured their support for the project and established the contact to their aforementioned Spanish colleagues, which I would like to sincerely thank them for. Especially I would like to mention my advisor Dr.-Ing. Christiane Förster who guided the project at my home university and whose advice helped a lot in finishing this thesis.

Aparte de la asistencia profesional, quiero dar las más afectuosas gracias a mis colegas de investigación como también a mis compañeros de piso por un tiempo agradable y divertido en Cataluña así como por una integración que no hubiera podido ser mejor.

En outre, j'aimerais remercier à tout ceux qui m'ont accompagné pendant ma formation et la connaissance desquels j'ai pu faire au cours de mes études. Soit à la Technische

Universität München soit à l'École Centrale Paris dans le cadre du programme double-diplôme TIME (Top Industrial Managers for Europe), merci beaucoup pour les moments merveilleux passés ensemble ainsi que les expériences extrêmement enrichissantes.

Darüber hinaus fühle ich mich überglücklich, so viele gute Freunde zu Hause zu haben, die den Kontakt auch in der Ferne nie haben abreißen lassen und mir die Rückkehr immer wieder zu einem einzigartigen Erlebnis gemacht haben.

Der größte Dank gilt jedoch mit Sicherheit meiner Familie – in ganz besonderem Maße meinen Eltern und meinem Bruder, die mich während meines kompletten Studiums bedingungslos unterstützt und mir stets den nötigen Rückhalt gegeben haben. Ohne sie wäre vieles nicht in der Art möglich gewesen, so dass ich letztendlich auch nicht hier stünde.

Abstract

This diploma thesis deals with the implementation of a fluid solver for incompressible and compressible flows within the multi-physics framework Kratos. The presentation of this environment based on the finite element method (FEM) and an introduction to multi-disciplinary problems in general are the starting point of this work and help understanding the following steps more easily.

Originating from the basic conservation equations for mass, momentum and energy, the Euler equations for inviscid flow are derived. In this context some approximations are presented that avoid the solution of the energy equation and allow the use of a general approach for the simulation of incompressible, slightly compressible and barotropic flow.
The implementation of the incompressible transient Navier-Stokes solver is described. To avoid the continuous problem, a fractional step scheme is presented in order to uncouple pressure and velocity components by a split of the momentum equation. Emphasis is placed on the nodal implementation using an edge-based data structure. Moreover, the orthogonal subscale stabilization – necessary because of the finite element discretization – is explained very briefly.
Subsequently, the solver is extended to compressible regime mentioning the respective modifications. For validation purposes numerical examples of incompressible and compressible flows in two and three dimensions round off this first part.

In a second step, the implemented flow solver is prepared for the fluid-structure coupling. After presenting solving procedures for multi-disciplinary problems, the arbitrary Lagrangian Eulerian (ALE) formulation is introduced and the conservation equations are modified accordingly.
Some preliminary tests are performed, particularly with regard to mesh motion and adjustment of the boundary conditions. Finally, expectations for the envisaged fluid-structure coupling are brought forward.

Contents

List of Figures

1 Introduction

1.1. Multi-Physics in general...

A wide range of scientific and engineering tasks possess an interdisciplinary character. This means multiple physical models are underlying and have to be considered simultaneously in order to simulate the temporal development of a certain phenomenon adequately. Well-known examples are thermal stress, electromechanic interaction, fluid-structure interaction, fluid flow with heat transport and chemical reactions, electromagnetic fluids (magnetohydrodynamics or plasma) and electromagnetically induced heating.

These multi-physics problems typically involve solving coupled systems of partial differential equations. Considering industrial applications with a considerable number of degrees of freedom, one can imagine that there are very high requirements for cpu power as well as for memory allocation. Although computer performance has been continuously increasing during the last decade, the vision of real-time aircraft or Formula One race car simulations is still far away. Tradeoffs have to be made between computational time and accuracy of the obtained results. In this context parallelization and the use of reduced order models play a decisive role to decrease the computational effort.

1.2. ... and Fluid-Structure Interaction in particular

A siginificant subcategory is the already mentioned fluid-structure interaction (FSI) that will be partially focused on in this work. It occurs when a fluid interacts with a solid structure, exerting pressure on it which may cause deformation in the structure and thus alter the flow of the fluid itself. Such interactions may be stable or oscillatory and are a crucial consideration in the design of many engineering systems.
Failing to consider the effects of FSI can be catastrophic, especially in large scale structures and those comprising materials susceptible to fatigue. The Tacoma Narrows suspension bridge is probably one of the most infamous examples of large-scale failure, the aeroelastic phenomenon of flutter on aircraft wings another one.

Aside from its destructive potential, FSI is responsible for countless useful effects in engineering. It allows fans and propellers to function; sails on marine vehicles to provide thrust; aerofoils on racecars to produce downforce, and our lungs to inflate when we breathe.

In general, the program to simulate FSI problems involves a structural and a fluid solver that are coupled by a certain interface in order to exchange variables and parameters. In the case of Kratos – a multi-disciplinary framework based on the finite element method – very potent modules for static and dynamic analysis of structures have already been implemented. Consequently, the aim of this work was the development of a fluid solver, capable of handling incompressible and compressible flows (as far as the sound barrier is not exceeded), and the preparation of the application for fluid-structure coupling.

1.3. Organization of the Document

This diploma thesis describes the implementation of an algorithm for incompressible and compressible flows in subsonic regime within a multi-disciplinary FEM framework. Afterwards, this fluid solver is prepared for the coupling with existing structural applications. Following this idea, the layout of the document is organized as follows:

Chapter 2 classifies multi-disciplinary problems and gives an overview of Kratos as finite element framework for such a type of simulations.

Chapter 3 details the implementation of a general algorithm for incompressible and compressible flows within Kratos, focussing on the construction of an edge-based data structure and defining a starting point on parallelization issues. Numerical examples are given for validation purposes.

Chapter 4 describes modifications of the fluid solver due to the arbitrary Lagrangian Eulerian formulation of the equations of motion, which are necessary for preparing the fluid-structure coupling. Moreover, basic tests with moving meshes are performed.

Chapter 5 recapitulates the implementation of the algorithm as well as the obtained results in order to draw conclusions and to hint at future spheres of action.

2 KRATOS – Multi-Physics FEM Environment

Based on the definition of multi-disciplinary problems and on their classification, essential background information on the structure of the finite element framework Kratos is given. This overview is necessary to understand the implementation of applications and the coupling of flow and structural solvers in the following chapters.
Beyond that, its interface with the pre- and post-processor GiD and the handling of simulation runs by Python scripts are presented.

2.1. Multi-Disciplinary Problems

Since the objective of this work is the extension of Kratos, the purpose of this environment shall be focused on first, that is to say the tackling *multi-disciplinary problems*. Different definitions exist for this term, but usually searching for a multi-disciplinary solution is referred to as solving a *coupled system* of different physical models together – a collection of dependent problems put together building up a complex model. Nevertheless, a more general definition shall be used here: solving a model which consists of components with different formulations and algorithms interacting together. It is important to mention that this difference may come not only from the different physical nature of the problems but also from their type of mathematical modeling or discretization.
A *field* is a subsystem of a multi-disciplinary model characterized by certain mathematical equations and conditions. More precisely a fluid field is considered in the following chapters, building up the FSI problem together with a structure field. Accordingly a *domain* is the part of the modeled space governed by the respective field equation.

The definition given above includes a variety of problems, each of them with its proper characteristics. Different classifications are possible to categorize them, reflecting e.g. the kind of interaction between subsystems or the type of domain interfaces.
As these aspects were important in the design of Kratos (Dadvand, 2007), they are illustrated in the following.

2.1.1. Weak and Strong Coupling

Considering a simple multi-disciplinary problem with two interacting subsystems S_1 and S_2 as shown in Figure 2.1. The calculation of the respective solutions $u_1(t)$ and $u_2(t)$ under applied forces $F(t)$ is up to the type of dependency between the subsystems.

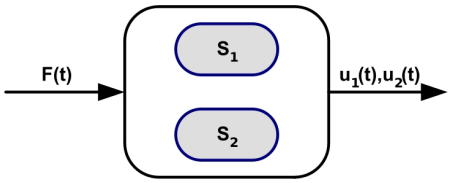


Figure 2.1: A general multi-disciplinary problem consisting of two subsystems

Weak Coupling Only one domain depends on the other one, which can be solved independently. That is why this type is also called *one-way* coupling. A thermal-structure problem is a good example where the material's property of the structure depends on the temperature while the thermal field can be solved independently, assuming that the temperature change due to structural deformation is very small. Figure 2.2 shows this type of coupling.

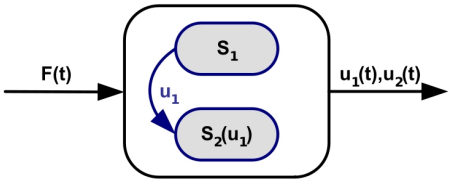


Figure 2.2: Example of a weakly coupled system

Strong Coupling Each subsystem depends on the other one, ruling out the seperate solution. Hence this type is also referred to as *two-way* coupling. The fluid-structure interaction problems for structures with large deformations fall into this category. The structural deformation is caused by the pressure resulting from the fluid flow on the one hand, whereas velocity and pressure of the fluid depend on the shape of the deformed structure on the other hand. Figure 2.3 shows this type of coupling.

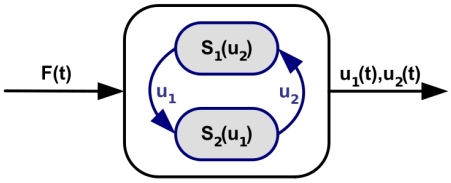


Figure 2.3: Example of a strongly coupled system

2.1.2. Interaction over Boundary and Domain

Apart from the type of dependency the classification also may be done on where the different subsystems interact with each other.

Interaction over Boundary In this category the interaction occurs at the domain boundaries. In a fluid-structure interaction problem as shown in Figure 2.4 the coupling of the two subsystems appears only on the boundary faces whereas interior points of each subsystem are not affected directly.

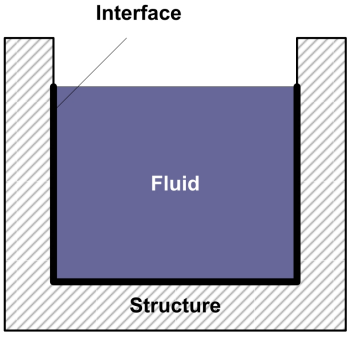


Figure 2.4: Boundary interaction in a fluid-structure problem

Interaction over Domain This category includes problems where domains can overlap totally or partially. Combustion processes or the thermal-fluid problem illustrated in Figure 2.5 are good examples. In the heating pipe the thermal domain overlaps the fluid domain.

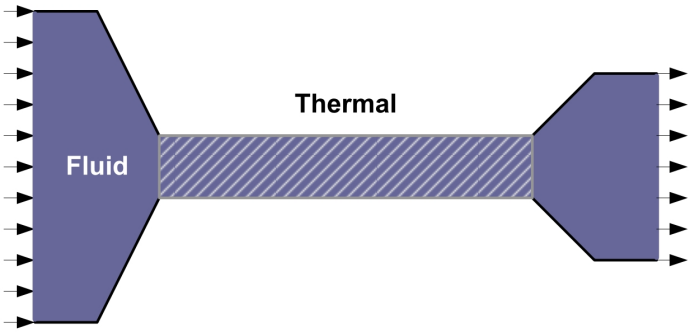


Figure 2.5: Domain interaction in a thermal-fluid problem

2.2. General Structure of Kratos

Kratos is an open source C++ framework to perform multi-disciplinary simulations based on the Finite Element Method (FEM). Therefore it provides several tools for easy imple-

mentation of finite element applications as well as a common platform for natural interaction of these modules in different ways. Kratos has been set up at the International Center for Numerical Methods in Engineering (CIMNE) in Barcelona and is currently being enhanced further, this work representing one of the recent extensions.

In order to enable the implementation of different sets of algorithms and formulations within this context, a general approach has been followed during its design process providing the necessary flexibility and extensibility (Dadvand, 2007). As a result, Kratos adresses itself to a variety of users ranging from developers (finite element experts as well as application developers) to engineers and designers using the package as a whole without getting involved in the programming.

2.2.1. Object-Oriented Approach

The main goal of an *object-oriented* structure is to split the whole problem into several objects and to define their interfaces. With regard to the simulation of multi-disciplinary problems using FEM, the objects defined in Kratos are based on a general finite element methodology. Figure 2.6 illustrates the main classes.

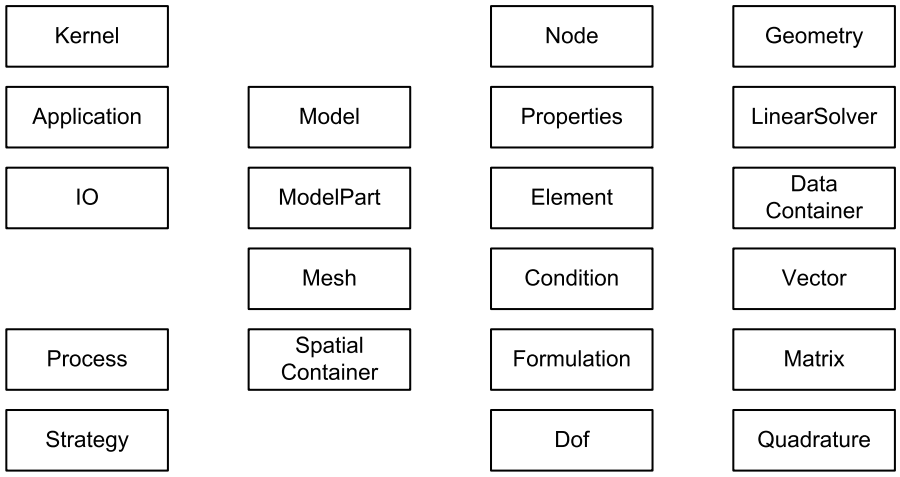


Figure 2.6: Main classes defined in Kratos

`Vector`, `Matrix` and `Quadrature` are designed by basic numerical concepts. `Node`, `Element`, `Condition` and `Dof` are defined directly from finite element concepts. `Model`, `Mesh` and `Properties` are coming from practical methodology used in finite element modeling completed by `ModelPart` and `SpatialContainer` for a better organization of all necessary analysis data. `IO`, `LinearSolver`, `Process` and `Strategy` are representing the different steps of finite element program flow. Finally, `Kernel` and `Application` handle the library management and define Kratos' interface.

2.2.2. Multi-Layer Design

Kratos uses a *multi-layer* approach, in which each object only interfaces with other objects in its layer or in layers below this one. Thereby dependencies inside the program get reduced, helping in the maintenance of the code on the one hand and clarifying the tasks for developers on the other hand. Figure 2.7 shows the multi-layer nature of Kratos being geared to the various user groups. For a better understanding the individual layers are presented hereafter using a bottom-up approach:

Basic Tools Layer holds all basic tools used in Kratos, that is mathematical definitions, solving procedures and build-up of data structures. In order to maximize their performance, advanced techniques in C++ are essential in this layer.

Base Finite Element Layer contains the ingredients that are necessary to implement a finite element formulation. The objects `Element`, `Node`, `Properties`, `Condition` and `Degrees of freedom` are defined here and in a manner of speaking hidden from the finite element developers.

Finite Element Layer is restricted to the basic and average features of language and uses the two layers below to optimize the performance without entering into details.

Data Structures Layer contains all objects organizing the data structure. This layer will be affected by the nodal based implementation requiring an edge-based data structure in compressed sparse row format.

Base Algorithms Layer provides the components building the extendible structure for algorithms.

User's Algorithms Layer contains all classes implementing the different algorithms in Kratos. In this layer the general algorithm for compressible and incompressible flows will be placed.

Applications' Interface Layer holds all objects that manage Kratos and its relation with other applications. Within this scope the new fluid application using the foremen-tioned algorithm will be defined.

Applications Layer contains the interface of certain applications with Kratos.

Scripts Layer provides a set of input/output scripts that can be used to activate respectively deactivate certain functionalities or to implement different algorithms from outside Kratos. As Python scripts have been used to handle our simulation runs, an example is given in the following section.

2.2.3. Python Interface

The use of Python start scripts for simulations has proven to be very convenient as it allows the user to adapt the program to his special needs, which is extremely useful during debugging an application or problem solving. Furthermore, nearly any parameter (e.g. tolerance values, initial and boundary conditions, etc.) can be changed "on the fly" without having to recompile the whole C++ source code.

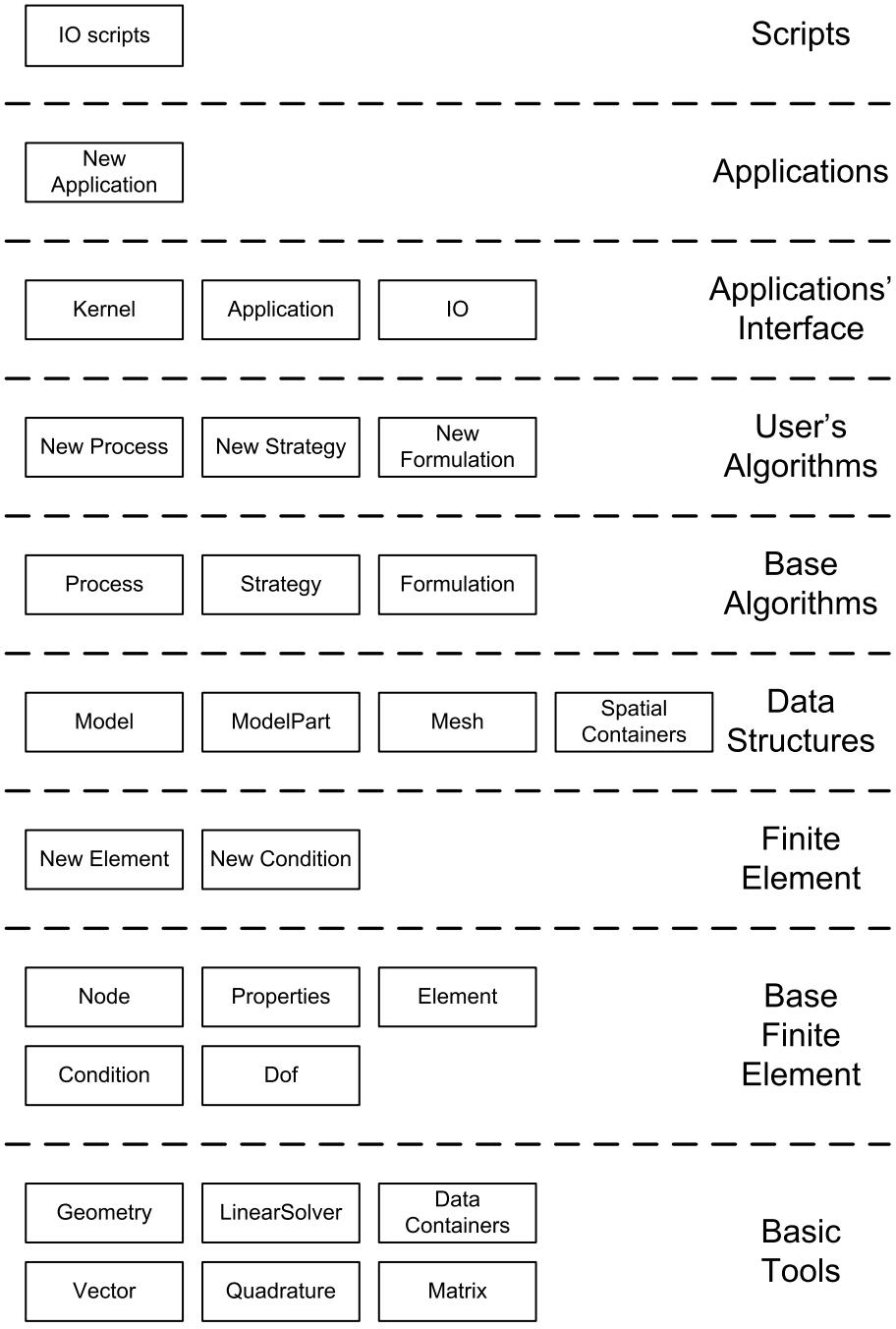


Figure 2.7: Division of Kratos' structure into layers

Whereas a complete Python script for a flow simulation on a moving mesh is appended to this document (A), the most relevant sections are given below in order to illustrate the mentioned advantages:

Listing 2.1: Easy access to simulation parameters by a Python start script

```
64 ######################################################################
65 # settings to be changed...
66
67 #INITIALIZE FLUID DOMAIN
68 for node in model_part.Nodes:
69   #change properties and initial values
70   initial_velocity = 1.0
71   node.SetSolutionStepValue(VELOCITY_X,0,initial_velocity)
72   node.SetSolutionStepValue(VELOCITY_Y,0,0.0)
73   node.SetSolutionStepValue(VELOCITY_Z,0,0.0)
74   pressure = 20 - (0.1 * node.X)
75   node.SetSolutionStepValue(PRESSURE,0,pressure)
76
77 #CHANGE BOUNDARY CONDITIONS
78 #boundary flags:
79 #1 - Velocity Inlet (Dirichlet)
80 #2 - No-Slip Condition (Dirichlet)
81 #3 - Slip Condition (Dirichlet)
82 #4 - Pressure & Slip Node
83 #5 - Pressure Inlet/Outlet (Neumann)
84 for node in model_part.Nodes:
85   #change no-slip condition to slip boundary
86   if(node.GetSolutionStepValue(IS_BOUNDARY) == 2.0)
87     node.SetSolutionStepValue(IS_BOUNDARY) == 3.0
88   #set boundary values
89   if(node.GetSolutionStepValue(IS_BOUNDARY) == 1.0):
90     inlet_velocity = 10.0
91     node.SetSolutionStepValue(VELOCITY_X,0,inlet_velocity)
92   if(node.GetSolutionStepValue(IS_BOUNDARY) == 4.0 or node.
       GetSolutionStepValue(IS_BOUNDARY) == 5.0):
93     pressure_outlet = node.GetSolutionStepValue(PRESSURE)
94     node.SetSolutionStepValue(EXTERNAL_PRESSURE,0,pressure_outlet)
95
96 #SET FURTHER VALUES
97 #time step size and output interval
98 dt = 0.01
99 n_steps= 1000
100 out = 1
101 output_step = 10
102 #stop criteria for iteration
103 tolerance = 1e-3
104 abs_tol = 1e-6
105 n_it_max = 100
106
107 # ...all settings defined
108 ######################################################################
```

2.3. GiD Pre- and Postprocessor

To perform the above-named multi-disciplinary simulations, input data describing the considered model – its geometry, its material properties as well as basic conditions and parameters – have to be defined. For this purpose GiD, the universal pre- and postprocessor developed and distributed by CIMNE, has been used. Providing the user with a graphical interface, it has been designed as an adaptive tool for geometrical modelling, data input and visualisation of results for all types of numerical simulation programs.

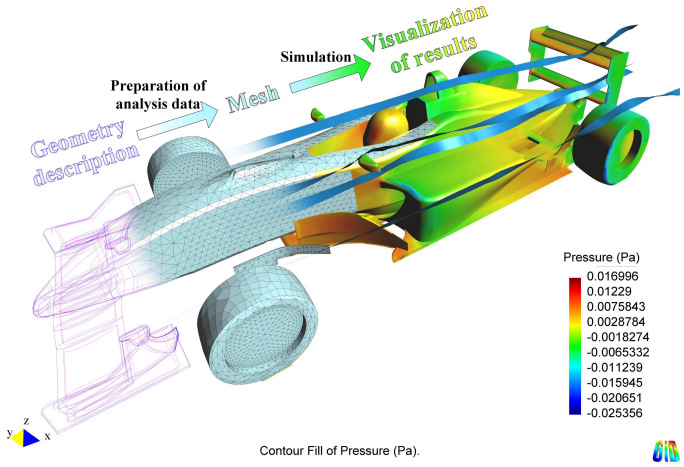


Figure 2.8: Geometry definition, mesh generation and visualization of results in GiD

Figure 2.8 illustrates the different steps surrounding the proper simulation process:

Geometry description - Apart from defining the two- respectively three-dimensional geometry of the model manually by points, lines and surfaces, multiple data formats from standard Computer Aided Design (CAD) software tools can be imported.

Mesh generation - Having assigned quality and spacing criteria of the mesh to geometrical entities, the user can dispose of various options. He has the choice between structured meshes for linear and quadratic elements, automatically generated unstructered meshes and semi-structured volume meshes (structured in one direction), including triangular, quadrilateral, hexahedral, prism and tetrahedral elements. Further editing utilities like mesh refinement by splitting elements, edge collapses and smoothing are also available.

Visualization of results - Once the simulation has been run, results can be visualized by all kinds of graphs: counter and vector plots, deformed geometry shapes, isosurfaces and stream lines to name just the most popular ones. Moreover animated sequences can be recorded and time relevant data can be extracted to perform further operations like the fast Fourier transform (FFT). Once more, data exchange with common post-processing tools like STL, NASTRAN and TECPLOT is possible.

One of GiD's main objectives is its adaptive character willing to provide a compatible interface to any kind of numerical simulation program, independant of the employed code (finite element, finite volume, finite difference, meshless). To successfully tackle this requirement, a so-called *problemtype* has to be designed for each application, defining elective properties and conditions as well as format conversion options. Up to now, such problemtypes exist for applications in solid and structural mechanics, fluid dynamics, heat transfer, electromagnetics and geomechanics.

2.3.1. Preparation of the Model

First of all, the geometry of the model is defined by points, lines, surfaces and volumes. Then a material is selected and the characteristics of domain and its boundaries have to be defined. For the implemented flow solver it is crucial to choose only the elements `Fluid2D` and `Fluid3D` for the two- respectively three-dimensional domain under "Fluid Element Type". Forthermore the boundary conditions `Condition2D` and `Condition3D` under "Fluid Boundary Condition" have to be set. As the boundary generally is divided into several distinctive parts, the Kratos flag variable `IS_BOUNDARY` is used to indicate

- **0** interior points,
- **1** a velocity inlet,
- **2** no-slip conditions,
- **3** slip conditions,
- **4** slip/pressure nodes and
- **5** a pressure outlet.

Figure 2.9 shows an example for the definition of the boundary flag that will be used within the algorithm to differentiate between the respective boundary types. Be careful with the corners of the domain as it might be necessary to set the value for the respective corner points separately.

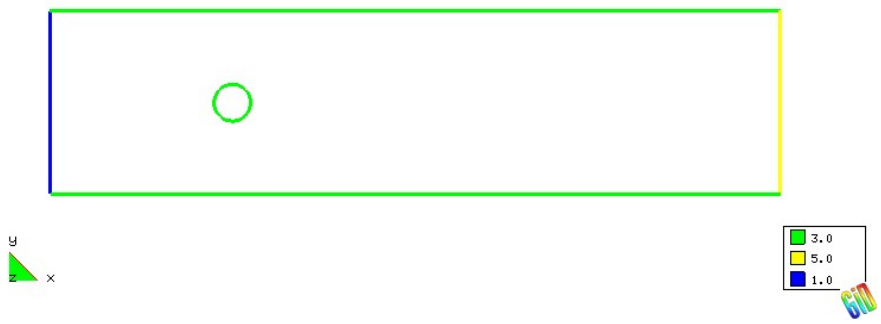


Figure 2.9: Definition of boundary conditions in GiD

Finally, the domain is meshed and the calculation files necessary to start the simulation run are written.

3

Fluid Solver – Implementation and Validation

In this chapter the implementation of a flow solver within the multi-physics code KRATOS is exposed step-by-step.
Starting with the derivation of the Euler equations for incompressible flow, the finite element method is applied for spatial discretization. Using a fractional step scheme, pressure and velocity components are uncoupled.
To avoid certain redundancies of information, many simple element-based flow solvers are suffering from (due to the cost of indirect addressing operations), an edge-based data structure has been chosen. Thus, the attention of one section will be directed to the necessary changes in storage and access of edge-based data as well as to the calculation of edge contributions to the global system matrices.
Furthermore the algorithm is expanded to compressible flow in subsonic regime. Finally, for validation purpose, numerical flow simulations of two- and three-dimensional test cases are performed.

3.1. Motivation

Introduced in the late 1950s in the aircraft industry, the finite element method has emerged as one of the most powerful numerical methods so far devised. Its main assets, having led to widespread acceptance and popularity, are

- the ease in modeling complex geometries,
- the consistent treatment of differential type boundary conditions and
- the possibility to be programmed in a flexible and general purpose format.

Standard finite element approximations are based upon the Galerkin formulation of the method of weighted residuals. This formulation has proven eminently successful in application to problems in solid/structural mechanics and in other situations, such as heat conduction, which is governed by a diffusion-type equation. This can be explained by the fact that, when applied to problems governed by self-adjoint elliptic or parabolic partial

differential equations, the Galerkin finite element method leads to symmetric stiffness matrices. In this case the difference between the finite element solution and the exact solution is minimized with respect to the energy norm.
However, in the case of fluid flow problems based on kinematical descriptions other than Lagrangian, non-symmetric convection operators appear in the formulation and thus the best approximation property in the energy norm of the Galerkin method is lost when convection dominates the transport process. In practice, Galerkin solutions to these problems are often corrupted by spurious node-to-node oscillations. As severe mesh and time step refinement clearly would undermine the practical utility of the method, stabilization techniques have to be applied. Moreover, in truly transient situations, space-time coupling is particularly crucial due to the directional character of propagation of information in hyperbolic problems (Donea and Huerta, 2003).

3.2. Governing Equations in Fluid Dynamics

The motion of fluid substances such as gases and liquids is determined by the **Navier-Stokes equations** named after Claude-Louis Navier and George-Gabriel Stokes. They represent a set of *non-linear partial differential equations* establishing a relation among the rates of change of velocity and pressure. Strictly spoken they only state the conservation of momentum so that, depending on the flow properties, further conservation laws for mass and energy are necessary to fully describe the motion.
In addition boundary and initial conditions have to be prescribed adequately in order to close the *initial boundary value problem*. It can be distinguished between

1. **Dirichlet**, prescribing the value of the unknown function,
2. **Neumann**, imposing the normal gradient, and
3. **Robin**, prescribing a combination of the unknown function and its gradient why this type is often refered to as mixed boundary condition.

3.2.1. Basic Conservation Equations

Mass Conservation

The conservation of mass contained in a material volume (a volume permanently containing the same particles of the continuum under consideration) is a fundamental law of Newtonian mechanics. According to Donea and Huerta (2003) it can be written as

$$\frac{dM}{dt} = \frac{d}{dt}\int_{V_m(t)} \rho dV = 0, \tag{3.1}$$

where the mass M is expressed by a volume integral of the fluid density ρ.
The material time derivative of the integral of a scalar function $f(\boldsymbol{x}, t)$ over the time-varying

material volume $V_m(t)$ is given by the *Reynolds transport theorem*:

$$\frac{d}{dt}\int_{V_m(t)} f(\boldsymbol{x},t)dV = \int_{V_c\equiv V_m(t)} \frac{\partial f(\boldsymbol{x},t)}{\partial t}dV + \int_{A_c\equiv A_m(t)} f(\boldsymbol{x},t)\boldsymbol{u}\cdot\boldsymbol{n}dA, \tag{3.2}$$

which holds for smooth functions $f(\boldsymbol{x},t)$. The volume integral on the right-hand side is defined over a control volume V_c (fixed in space) coinciding with the moving material volume $V_m(t)$ at the considered instant t in time. Similarly, the fixed control surface A_c coincides at time t with the closed surface $A_m(t)$ bounding the material volume $V_m(t)$. In the surface integral, $\boldsymbol{u}$ denotes the material velocity of points on the boundary $A_m(t)$ whereas $\boldsymbol{n}$ is the unit outward normal to the surface $A_m(t)$ at the considered instant.
Applied to the law of mass conservation, equation (3.1) can be rewritten in the following form:

$$\frac{dM}{dt} = \int_{V_m(t)} \frac{\partial\rho}{\partial t}dV + \int_{A_m(t)} \rho\boldsymbol{u}\cdot\boldsymbol{n}dA = \int_{V_m(t)} \left(\frac{\partial\rho}{\partial t} + \boldsymbol{\nabla}\cdot(\rho\boldsymbol{u})\right)dV = 0. \tag{3.3}$$

Since this relation is independent of the choice of the volume $V_m(t)$, the integrand must be identically zero. Hence

$$\frac{\partial\rho}{\partial t} + \boldsymbol{\nabla}\cdot(\rho\boldsymbol{u}) = 0 \tag{3.4a}$$

at all points in the fluid. A different form of this equation is obtained by expanding the divergence term and considering that two of the terms together make up the material derivative of the density.

$$\frac{d\rho}{dt} + \rho\boldsymbol{\nabla}\cdot\boldsymbol{u} = 0 \tag{3.4b}$$

Apart from *mass-conservation equation* equation (3.4) is also termed *continuity equation.*

Momentum Conservation

The *momentum equation* can be derived from Newton's second law (Wikipedia – The Free Encyclopedia):

> *"The alteration of motion is ever proportional to the motive force impressed, and is made in the direction of the right line in which that force is impressed."*

Applied to flows this statement reads that changes in momentum in a chosen volume of fluid are equal to the sum of all forces $\boldsymbol{F}$ acting on this selected volume, including for example dissipative viscous forces (similar to friction), changes in pressure, gravity and other forces acting inside the fluid (Candel, 2005)

$$\frac{d}{dt}\int_{V_m(t)} \rho\boldsymbol{u}dV = \boldsymbol{F}. \tag{3.5}$$

In general, a portion of fluid is acted upon by both volume and surface forces:

1. Denoting by $\boldsymbol{g}$ the volume force per unit mass of fluid, the total volume force on the selected portion of fluid is
$$\int_{V_m(t)} \rho \boldsymbol{g} dV. \tag{3.6a}$$

2. The k-component of the surface force exerted across a surface element of area dA and normal $\boldsymbol{n}$ is given by $\sigma_{kl} n_l dA$ – using the summation convention on repeated indices – so that the total force exerted on the selected portion of fluid by the surrounding matter can be expressed in terms of the Cauchy stress as
$$\int_{A_m(t)} \sigma_{kl} n_l dA = \int_{V_m(t)} \frac{\partial \sigma_{kl}}{\partial x_l} dV \quad \text{or} \quad \int_{A_m(t)} \boldsymbol{\sigma} \cdot \boldsymbol{n} dA = \int_{V_m(t)} \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} dV. \tag{3.6b}$$
From a physical point of view, the symmetric stress tensor $\boldsymbol{\sigma}$ can be divided up into two parts once more:
$$\sigma_{kl} = -p\delta_{kl} + \tau_{kl} \quad \text{or} \quad \boldsymbol{\sigma} = -p\boldsymbol{I} + \boldsymbol{\tau}$$
   a) a *mean hydrostatic stress tensor* $-p\boldsymbol{I}$, which tends to change the volume of the fluid in an isotropic manner and only depends on its thermodynamic state, and
   b) a deviatoric component called the *stress deviator tensor* $\boldsymbol{\tau}$, which tends to distort the fluid and is essentially linked to its state of deformation.

Introducing equations (3.6) into (3.5) yields
$$\frac{d}{dt} \int_{V_m(t)} \rho \boldsymbol{u} dV = \int_{V_m(t)} (\rho \boldsymbol{g} + \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}) \, dV. \tag{3.7}$$

Making use of the Reynolds transport theorem again (this time in vector form), the left-hand side of equation (3.7), that is to say the momentum for the portion of fluid of volume $V_m(t)$ enclosed by the material surface $A_m(t)$, can be expressed as
$$\begin{aligned} \frac{d}{dt} \int_{V_m(t)} \rho \boldsymbol{u} dV &= \int_{V_m(t)} \frac{\partial (\rho \boldsymbol{u})}{\partial t} dV + \int_{A_m(t)} (\rho \boldsymbol{u} \otimes \boldsymbol{u}) \cdot \boldsymbol{n} dA \\ &= \int_{V_m(t)} \left(\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) \right) dV. \end{aligned} \tag{3.8}$$

The symbol $\otimes$ denoting the tensor product, the term $\rho \boldsymbol{u} \otimes \boldsymbol{u}$ leads to another second-rank tensor $\boldsymbol{T} = \rho[u_l u_k]$ with $l, k = 1, ..., n_{dim}$, whose divergence can be calculated componentwise
$$[\boldsymbol{\nabla} \cdot \boldsymbol{T}]_l := \sum_{k=1}^{n_{dim}} \frac{\partial T_{lk}}{\partial x_k} \quad \text{for} \quad l = 1, ..., n_{dim}.$$

In this work we will use the indices k and l to indicate space dimensions, whereas i and j are reserved for nodal values and shape functions. Furthermore the summation convention is applied whenever repeated indices appear.

Equating the right-hand sides of equations (3.7) and (3.8) writes the momentum balance for the selected material volume of fluid, which accounts for both previous actions:

$$\int_{V_m(t)} \left(\frac{\partial(\rho \boldsymbol{u})}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u}) \right) dV = \int_{V_m(t)} (\rho \boldsymbol{g} + \boldsymbol{\nabla} \cdot \boldsymbol{\sigma})\, dV. \tag{3.9}$$

This integral relation holds for all choices of the material volume $V_m(t)$ so that we finally obtain the so-called *equation of motion*:

$$\frac{\partial(\rho \boldsymbol{u})}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u} - \boldsymbol{\sigma}) = \rho \boldsymbol{g} \tag{3.10a}$$

Analog to the continuity equation (3.4) a second form of the *momentum-conservation equation* exists, this time incorporating the material time derivative of the velocity:

$$\rho \frac{d\boldsymbol{u}}{dt} - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = \rho \boldsymbol{g}. \tag{3.10b}$$

One can easily deduce the non-linear character of the momentum equation (3.10) due to the convective acceleration term $(\boldsymbol{u} \cdot \boldsymbol{\nabla})\boldsymbol{u}$ coming from linearization and describing the time independent acceleration of the fluid with respect to space. This represents a more than significant feature given that it does not only complicate the solution but also requires a special treatment applying the finite element method, as will be shown later on.

Energy Conservation and Equation of State

We note that velocity components u_k, pressure p and density ρ are the independant variables in equations (3.4) and (3.10). Obviously, there is one variable too many for this equation system to be capable of solution. However, if the density is assumed constant (as in incompressible flow) or if a single relationship linking pressure and density can be established (as in isothermal flow with small compressibility) the system becomes complete and is solvable.

More generally, the state variables pressure p, density ρ and absolute temperature T are related by an *equation of state* in the form of

$$\rho = \rho(p, T) \tag{3.11}$$

For a hypothetical ideal gas this takes the form

$$\rho = \frac{p}{RT} \quad \text{respectively} \quad p = \rho R T \tag{3.12}$$

where R is the specific gas constant of the medium. In such a general case, it is necessary to supplement the governing equation system by the equation of *energy conservation*. This equation is indeed of interest even if it is not coupled, as it provides additional information about the behaviour of the system.

According to (Zienkiewicz and Taylor, 2000), the total energy equation in terms of partial time derivative reads

$$\frac{\partial(\rho e)}{\partial t} + \boldsymbol{\nabla} \cdot \big((\rho e + p)\boldsymbol{u}\big) - \lambda \boldsymbol{\nabla}^2 T - q_H + \boldsymbol{\nabla} \cdot (\boldsymbol{\tau} \cdot \boldsymbol{u}) = \boldsymbol{u} \cdot \rho \boldsymbol{g} \tag{3.13}$$

where e is the total energy per unit mass of fluid and can be calculated as the sum of internal and kinetic energy. Apart from the classical mechanical energies, energy transfer due to conduction and chemical reactions have been taken into account as well as energy dissipation due to internal stresses. Accordingly λ is the isotropic thermal conductivity and q_H represents the heat source terms specified per unit volume. By the way, radiation generally is confined to boundaries.

Nevertheless, we will not immerse in the derivation of this equation as our main focus lies on a general approach for incompressible and compressible flows in subsonic regime where some simplifications are possible. First, a *subsonic flow* is characterized by

$$Ma < 1 \tag{3.14}$$

where the dimensionless *Mach number*

$$Ma = \frac{|\boldsymbol{u}|}{c} \tag{3.15}$$

is defined as the ratio of the module of the fluid velocity $\boldsymbol{u}$ and the positive quantity $c = \sqrt{\frac{dp}{d\rho}}$, known as the *speed of sound* in the medium. The Mach number, which is defined locally, gives an idea of compressibility of the flow at any given point. When incompressible flows are considered, density gradients are not related to pressure ones. In fact, density is taken as a constant. In this case, the speed of sound can be considered as a constant, much larger than the local convective velocity. On the other hand, in compressible flow it is a quantity that varies in space following changes in thermodynamic properties.

In this context we will introduce two approximations we will make use of later on (Vázquez et al., 1999). In the case of **slightly compressible flows** we will use the relation

$$\frac{d\rho}{dp} = \frac{1}{c^2} \tag{3.16}$$

to link density and pressure gradients by defining a constant value for the speed of sound. Note that the incompressible case is included implicitely by considering $c \to \infty$.
The second simplification concerns compressible barotropic flows, that is to say fluids for which there is an equation of state that involves only density and pressure, and not the temperature. In general we write this equation as $p = p(\rho)$, but we will particularize it to the case

$$p = A\rho^{\gamma} \tag{3.17}$$

where A and γ, the *adiabatic exponent*, are physical constants. This situation is found for example in the case of **isentropic flow of perfect gases** and leads to the following relation

$$\frac{d\rho}{dp} = \gamma A \rho^{\gamma - 1} = \frac{\gamma p}{\rho}. \tag{3.18}$$

3.2.2. Navier-Stokes Equations

The governing equations derived in the preceding sections can be written in the general conservative form

$$\frac{\partial \boldsymbol{V}}{\partial t} + \frac{\partial \boldsymbol{F}_k}{\partial x_k} + \frac{\partial \boldsymbol{G}_k}{\partial x_k} + \boldsymbol{Q} = \boldsymbol{0} \tag{3.19}$$

in which the conservation equations for mass, momentum and energy provide the particular entries to the vectors that presented below, for the sake of clarity once in indicial notation:

- the independent variable vector
$$\boldsymbol{V} = \begin{bmatrix} \rho \\ \rho u_1 \\ \rho u_2 \\ \rho u_3 \\ \rho e \end{bmatrix}, \tag{3.20a}$$
- the convective flux vector
$$\boldsymbol{F}_k = \begin{bmatrix} \rho u_k \\ \rho u_1 u_k + \delta_{1k} p \\ \rho u_2 u_k + \delta_{2k} p \\ \rho u_3 u_k + \delta_{3k} p \\ u_k(\rho e + p) \end{bmatrix}, \tag{3.20b}$$
- the diffusion flux vector
$$\boldsymbol{G}_k = \begin{bmatrix} 0 \\ -\tau_{1k} \\ -\tau_{2k} \\ -\tau_{3k} \\ -\tau_{kl} u_l - \lambda \frac{\partial T}{\partial x_k} \end{bmatrix} \tag{3.20c}$$
- and the source term vector
$$\boldsymbol{Q} = \begin{bmatrix} 0 \\ -\rho g_1 \\ -\rho g_2 \\ -\rho g_3 \\ -\rho g_l u_l - q_H \end{bmatrix}. \tag{3.20d}$$

3.2.3. Euler Equations

The effects of viscosity and heat conduction are important in the immediate vicinity of solid surfaces situated in the flow domain or at its boundaries. The region in which viscosity and conduction have to be taken into account is called *boundary layer* and represents a very thin layer around the respective contour. Outside of the boundary layer the flow may be considered as the one of an ideal fluid. An *ideal fluid* is defined as inviscid, the tensor of viscous forces disappearing in the conservation equations of the momentum.

Introducing the assumption of an inviscid fluid in the complete set of equations presented in Equation (3.19) and neglecting the influence of heat conduction, a particular case is obtained ($\boldsymbol{G} = \boldsymbol{0}$) – known as *Euler equations*:

$$\frac{\partial \boldsymbol{V}}{\partial t} + \frac{\partial \boldsymbol{F}_k}{\partial x_k} + \boldsymbol{Q} = \boldsymbol{0} \tag{3.21}$$

where the arrays $\boldsymbol{V}$, $\boldsymbol{F}_k$ and $\boldsymbol{Q}$ are defined as before. Table 3.1 rewrites explicitely the Euler equations for compressible flows assuming that no heat source is present ($q_H = 0$).

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{u}) = 0 \tag{3.22a}$$

$$\frac{\partial (\rho \boldsymbol{u})}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \boldsymbol{u} \otimes \boldsymbol{u} + p \boldsymbol{I}) = \rho \boldsymbol{g} \tag{3.22b}$$

$$\frac{\partial (\rho e)}{\partial t} + \boldsymbol{\nabla} \cdot \big((\rho e + p) \boldsymbol{u}\big) = \boldsymbol{u} \cdot \rho \boldsymbol{g} \tag{3.22c}$$

Table 3.1: Set of Euler equations for compressible flows

The above set is convenient and physically meaningful, defining the *conservation* of important quantities. Nevertheless, many alternative forms of the above equations are given in literature, obtained by combinations of the various equations. Doing so, one shall keep in mind that equations written in *non-conservative* form may yield incorrect, physically unmeaningful, results in problems where shock discontinuities are present.
It is correct that the main interest of this work is the study of subsonic flow, where this shortcoming could be accepted for the time being. However, in the prospect of future extension of the solver to supersonic regime, the conservative form will be used in order to avoid misunderstandings.

For the sake of simplicity, the implementation will start with a solver for incompressible flows. In continuum mechanics an *incompressible flow* is a solid or fluid flow in which the divergence of velocity $\boldsymbol{u}$ is zero. This is more precisely termed *isochoric* flow:

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0. \tag{3.23}$$

Note that isochoric as well as incompressible describe the flow and do not refer to material properties. However, using the incompressibility assumption in the continuity equation (3.4) claims that the mass density is constant following the material element.

$$\frac{d\rho}{dt} + \rho \boldsymbol{\nabla} \cdot \boldsymbol{u} = \frac{d\rho}{dt} = 0 \tag{3.24}$$

In this case of constant mass density, pressure and temperature are directly linked by the equation of state (3.12) so that no energy equation is needed (see Table 3.2).

$$\boldsymbol{\nabla} \cdot \boldsymbol{u} = 0 \tag{3.25a}$$

$$\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla}) \boldsymbol{u} + \frac{1}{\rho} \boldsymbol{\nabla} p = \boldsymbol{g} \tag{3.25b}$$

Table 3.2: Set of Euler equations for incompressible flows

3.3. Edge-Based Data Structure

The motivation for implementing the Euler equations presented above in an nodal based manner is twice. On the one hand, well-known properties of and experience with the finite element method can be used to full capacity. On the other hand, the use of an edge-based data structure does not only enforce global conservation and symmetry at discrete level, it also facilitates the matrix-vector multiplication by pre-calculating certain integrals.

3.3.1. Nodal Implementation

Considering a finite element approximation with shape-functions N_i, the typical formation of the right-hand side (RHS) requires the evaluation of integrals given by

$$\boldsymbol{r}_i = \int N_i \boldsymbol{r}(\boldsymbol{u}) d\Omega = \sum_{elem} \int N_i \boldsymbol{r}(N_j \boldsymbol{u}_j) d\Omega_{el}. \tag{3.26}$$

These integrals operate on two sets of data:

- *point-data* for $\boldsymbol{r}_i$ and $\boldsymbol{u}_i$, and
- *element-data* for volumes, shape functions and its derivatives.

The flow of information is as follows:

1. gather point information into the element (e.g. $\boldsymbol{u}_i$),
2. operate on element-data to evaluate the integral in equation (3.26) and
3. scatter-add element RHS data to point-data in order to obtain $\boldsymbol{r}_i$.

For many simple flow solvers operating on vector-machines, the effort in step 2 is be minor compared to the cost of indirect addressing operations in steps 1 and 3 (Löhner, 2001). This problem may be overcome for low-order elements by changing the element-based data structure into an edge-based one, which eliminates certain redundancies of information. These can be highly demanding in terms of cpu-time: a study in Soto et al. (2004) revealed that the FLOPs (floating point operations) overhead ratio between an element-based implementation and an edge-based one is approximately 2.5.
Moreover, a standard shared-memory parallelization of the elemental loop is complicated because the contributions to the matrix term ij come from more than one element. Hence, a kind of coloring algorithm would be necessary to avoid the simultaneous access of edge data ij by elements in different processors. By contrast the parallelization in the presented edge-based implementation is straight-forward: two nested loops are performed, the main loop – which is the one to parallelize – over the mesh points i and the inner one over its neighbours j, connected by the edges ij. The contributions of edge ij are computed only when the nodal point i is accessed so that no coloring algorithm is required. It is possible, particularly with regard to the symmetry of Laplacian-like terms, to benefit from this by storing only half of the values. Nevertheless, this has not been realized in this work. In order to facilitate the envisaged parallelization, edge ji (accessed only for the nodal point j) is considered different from edge ij.

The idea of a nodal implementation (Codina and Folch, 2004) with an edge-based data structure is to express all contributions in terms of

$$\int_\Omega N_i N_j \mathrm{d}\Omega, \quad \int_\Omega \frac{\partial N_i}{\partial x_k}\frac{\partial N_j}{\partial x_l}\mathrm{d}\Omega, \quad \int_\Omega N_i \frac{\partial N_j}{\partial x_k}\mathrm{d}\Omega \quad \text{and} \quad \int_\Omega \frac{\partial N_i}{\partial x_k} N_j \mathrm{d}\Omega. \tag{3.27}$$

These expressions can be derived from the element integrals known from Galerkin weighted residual approximations as illustrated in Figure 3.1 and will be referred to as the edge contributions of the mass matrix $\mathbf{M}$, the Laplacian operator $\mathbf{L}$, the gradient $\mathbf{G}$ and the transposed gradient $\mathbf{G}^T$ with its components m_{ij}, $l_{ij,kl}$, $g_{ij,k}$ and $g_{ji,k}$ respectively. For

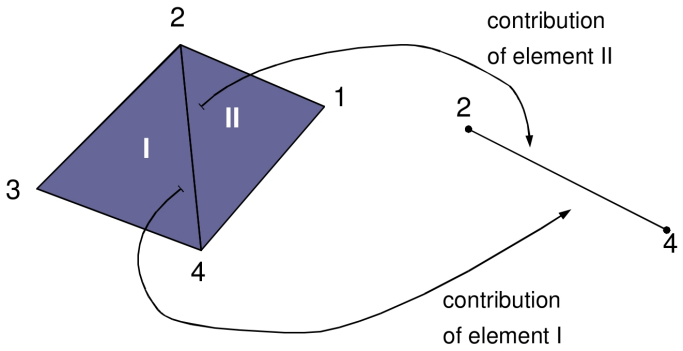


Figure 3.1: Build-up of edge contributions from element data

fixed domains, all the integrals in (3.27) can be computed a priori, that is to say at the beginning of the simulation run, and are then stored in a standard compressed-sparse-row format. Only if the domain is remeshed, the integrals will have to be re-computed.

Listing 3.1 shows the definition of the presented data structure within Kratos.

Listing 3.1: Definition of an edge-based data structure

```
1  namespace Kratos
2  {
3    //structure definition for use in CSR format
4    template<unsigned int TDim>
5    struct EdgesStructureType {
6      //consistent mass of edge ij
7      //(M = Ni * Nj * dOmega)
8      double Mass;
9      //components kl of the laplacian operator of edge ij
10     //(L = dNi/dxk * dNj/dxl * dOmega)
11     boost::numeric::ublas::bounded_matrix<double,TDim,TDim>
           LaplacianIJ;
12     //components k of the gradient of edge ij
13     //(G = Ni * dNj/dxk * dOmega)
14     array_1d<double, TDim> GradientJ;
15     //components k of the transposed gradient of edge ij
16     //(GT = dNi/dxk * Nj * dOmega)
17     array_1d<double, TDim> GradientI;
```

```
18    };
19  }
```

3.3.2. Compressed-Sparse-Row Format

The de facto standard storage format for unstructured sparse matrices is the *compressed-sparse-row* (CSR) format. The idea behind CSR is to pack each row by only storing the non-zero elements (White and Sadayappan, 1997). Obviously this turns out to be quite effective in an nodal based implementation since the contributions to the nodal term $\boldsymbol{r}_i$ are delivered only be the edges ij connection point i with its neighbours j.

$$\boldsymbol{A} = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 5 & 6 \\ 0 & 7 & 0 & 8 \end{pmatrix}$$

`mRowStartIndex[i_node]`

csr	0	2	3	6	8

`mColumnIndex[csr_index]`

j	0	1	1	0	2	3	1	3

`mEdgeValue[csr_index]`

a_{ij}	1	2	3	4	5	6	7	8

Table 3.3: Storage example for the compressed-sparse-row format

Considering the matrix $\boldsymbol{A}$ on the left-hand side of Table 3.3, one quickly will remark that each row may have its own structure. That is the reason way the start index of each row (`mRowStartIndex[i_node]`) as well as the column index of the non-zero entries (`mColumnIndex[csr_index]`) have to be stored both. Finally, a third unidimensional array is used to store the value a_{ij} (`mEdgeValue[csr_index]`). The right-hand side of Table 3.3 points out the CSR storage of matrix $\boldsymbol{A}$, where the CSR index hints at the position of column index and edge value in the two remaining one-dimensional arrays.

With a sparse matrix in CSR, matrix-vector multiplication can be implemented with a simple nested pair of loops. Listing 3.2 illustrates the loop over all edges in the implemented C++ source code. Whereas the outer loop is over the rows of matrix $\boldsymbol{A}$, the inner loop processes one non-zero row element after the other.

Listing 3.2: Loop over edge data in CSR storage

```
1  void DoEdgeLoop (ModelPart& rModelPart, MatrixContainer&
       rMatrixContainer)
2  {
3  KRATOS_TRY
4
5    //get number of nodes
6    unsigned int n_nodes = rModelPart.Nodes().size();
7
8    //loop over all nodes
9    for (unsigned int i_node = 0; i_node < n_nodes; i_node++)
10   {
11     //loop over neighbours j of node i
```

```
12      for (unsigned int csr_index=rMatrixContainer.GetRowStartIndex()
            [i_node]; csr_index!=rMatrixContainer.GetRowStartIndex()[
            i_node+1]; csr_index++)
13      {
14        //get global index of neighbouring node j
15        unsigned int j_neighbour = rMatrixContainer.GetColumnIndex()[
              csr_index];
16
17        //reference for mass component as an example of edge data
18        double& m_ij = rMatrixContainer.GetEdgeValues()[csr_index].
              Mass;
19        //reference for velocity as an example of point data
20        const array_1d<double, TDim>& u_j = mVelocity[j_neighbour];
21
22        //perform edge-based operations
23        [...]
24
25      }
26
27      //perform nodal based operations
28      [...]
29
30    }
31
32  KRATOS_CATCH("")
33  }
```

One of the core operations of iterative sparse solvers is sparse matrix-vector multiplication. A parallel implementation of this multiplication must maintain scalability in order to achieve high performance. This scalability depends on the balanced mapping of matrices and vectors among the distributed processors, on minimizing inter-processor communication and on a high single node performance.
A parallel implementation within Kratos, for instance by the bias of OpenMP, requires that the rows of the edge-based sparse matrices are distributed among the processors, with the rows local to a processor stored in CSR. Matrix-vector multiplication is then performed using an "owner computes" strategy.

Before using the presented loop over edges, the CSR data has to be computed. This is done by two functions `ConstructCSRVector` and `BuildCSRData`, whose important sections are shown by Listings B.1 and B.2 in the appendix.
In the first case a loop over all nodes (`i_node`) of the mesh is performed in order to determine their neighbours (`j_neighbour`). This step is necessary in order to define the structure of the CSR vector correctly. Besides, the nested loop over the neighbouring nodes is used to initialize the edge contributions with zero.
This initialization is quite convenient as in the build-up of the edge data integrals have to be summed up. Therefore a main loop over the elements has to be performed as demonstrated in Listing B.2. Inside this loop two further loops over the nodes of the considered element are nested in order to assign the element contribution to the respective edges. Note that no "diagonal entries" are stored: on the one hand this would be weird as

no edge ii exists, and on the other hand they are calculated when necessary by enforcing conservation properties.

In the following two subsections the geometrical expressions required while going from an element-based data structure to an edge-based data structure will be derived according to Löhner (2001).

3.3.3. Laplacian Operator

In the case of the Laplace operator, the RHS in the domain yields on the basis of equation (3.26)

$$\boldsymbol{r}_i = -\int_\Omega \nabla N_i \cdot \nabla N_j \mathrm{d}\Omega \boldsymbol{u}_j = -\left[\sum_{elem} \int_\Omega \nabla N_i \cdot \nabla N_j \mathrm{d}\Omega\right] \boldsymbol{u}_j \tag{3.28}$$

This integral can be split into those shape-functions that are the same as N_i and those that are different ($j \neq i$):

$$\boldsymbol{r}_i = -\sum_{j\neq i} \left[\sum_{elem} \int_\Omega \nabla N_i \cdot \nabla N_j \mathrm{d}\Omega\right] \boldsymbol{u}_j - \sum_{elem} \int_\Omega \nabla N_i \cdot \nabla N_i \mathrm{d}\Omega \boldsymbol{u}_i \tag{3.29}$$

Introducing the conservation property of the shape-function derivatives

$$\frac{\partial N_i}{\partial x_k} = -\sum_{j\neq i} \frac{\partial N_j}{\partial x_k} \tag{3.30}$$

the second term of the right-hand side may be rewritten differently

$$\boldsymbol{r}_i = -\sum_{j\neq i} \left[\sum_{elem} \int_\Omega \nabla N_i \cdot \nabla N_j \mathrm{d}\Omega\right] \boldsymbol{u}_j + \left[\sum_{elem} \int_\Omega \nabla N_i \cdot \sum_{j\neq i} \nabla N_j \mathrm{d}\Omega\right] \boldsymbol{u}_i \tag{3.31}$$

or, after interchange of the double sums,

$$\boldsymbol{r}_i = \sum_{j\neq i} l_{ij}(\boldsymbol{u}_i - \boldsymbol{u}_j) \quad \text{with} \quad l_{ij} = \sum_{elem} \int_\Omega \nabla N_i \cdot \nabla N_j \mathrm{d}\Omega. \tag{3.32}$$

It can be observed that a change in indices (ij versus ji) leads to $l_{ij} = l_{ji}$, which is expected from the symmetry of the Laplace operator.

3.3.4. Gradient and Transposed Gradient

We now proceed to first derivatives, the Euler fluxes being a typical example. The RHS is given by an expression of the form

$$\boldsymbol{r}_i = -\int_\Omega N_i \frac{\partial N_j}{\partial x_k} \mathrm{d}\Omega \boldsymbol{F}_{j,k} \tag{3.33}$$

where $\boldsymbol{F}_{j,k}$ denotes the flux in the k-th dimension at node j. This integral is again separated into shape-functions that are not equal to N_i and those that are equal

$$\boldsymbol{r}_i = -\sum_{j\neq i}\left[\sum_{elem}\int_\Omega N_i\frac{\partial N_j}{\partial x_k}\mathrm{d}\Omega\right]\boldsymbol{F}_{j,k} - \sum_{elem}\int_\Omega N_i\frac{\partial N_i}{\partial x_k}\mathrm{d}\Omega\boldsymbol{F}_{i,k} \tag{3.34}$$

Once more the conservation property (3.30) is used to obtain

$$\boldsymbol{r}_i = -\sum_{j\neq i}\left[\sum_{elem}\int_\Omega N_i\frac{\partial N_j}{\partial x_k}\mathrm{d}\Omega\right]\boldsymbol{F}_{j,k} + \left[\sum_{elem}\int_\Omega N_i\sum_{j\neq i}\frac{\partial N_j}{\partial x_k}\mathrm{d}\Omega\right]\boldsymbol{F}_{i,k} \tag{3.35}$$

This may be restated as

$$\boldsymbol{r}_i = \sum_{j\neq i} g_{ij,k}(\boldsymbol{F}_{i,k} - \boldsymbol{F}_{j,k}) \quad \text{with} \quad g_{ij,k} = \sum_{elem}\int_\Omega N_i\frac{\partial N_j}{\partial x_k}\mathrm{d}\Omega, \tag{3.36}$$

which turns out to be very convenient as the gradient g_{ii} does not appear in the equations and thus no "diagonal entries" of the original element matrices have to be stored. This means that the whole process of assembly (see Listing B.2) can be performed within one loop over all the edges, whose contributions are stored one after another in the CSR vector.

When the variational formulation of the problem imposes a weak gradient

$$\boldsymbol{r}_i = -\int_\Omega \frac{\partial N_i}{\partial x_k}N_j\mathrm{d}\Omega\boldsymbol{F}_{j,k}, \tag{3.37}$$

for instance due to partial integration in order to impose boundary values, a similar path can be followed to derive the corresponding statement. After splitting up the integral relative to the shape-functions

$$\boldsymbol{r}_i = -\sum_{j\neq i}\left[\sum_{elem}\int_\Omega \frac{\partial N_i}{\partial x_k}N_j\mathrm{d}\Omega\right]\boldsymbol{F}_{j,k} - \sum_{elem}\int_\Omega \frac{\partial N_i}{\partial x_k}N_i\mathrm{d}\Omega\boldsymbol{F}_{i,k}, \tag{3.38}$$

the conservation property (3.30) of the shape-function derivatives is applied once more

$$\boldsymbol{r}_i = -\sum_{j\neq i}\left[\sum_{elem}\int_\Omega \frac{\partial N_i}{\partial x_k}N_j\mathrm{d}\Omega\right]\boldsymbol{F}_{j,k} + \sum_{elem}\int_\Omega\sum_{j\neq i}\frac{\partial N_j}{\partial x_k}N_i\mathrm{d}\Omega\boldsymbol{F}_{i,k}, \tag{3.39}$$

so that we finally obtain

$$\begin{aligned}\boldsymbol{r}_i = \sum_{j\neq i}(g_{ij,k}\boldsymbol{F}_{i,k} - g_{ji.k}\boldsymbol{F}_{j,k}) \quad &\text{with} \quad g_{ij,k} = \sum_{elem}\int_\Omega N_i\frac{\partial N_j}{\partial x_k}\mathrm{d}\Omega \\ &\text{and} \quad g_{ji,k} = \sum_{elem}\int_\Omega \frac{\partial N_i}{\partial x_k}N_j\mathrm{d}\Omega.\end{aligned} \tag{3.40}$$

This is the reason why we initially calculated and stored both gradients. We can conclude that a change in indices ij versus ji leads to the following relation

$$g_{ji,k} = -g_{ij,k} + \int_{\Gamma} N_j N_i n_k \mathrm{d}\Gamma, \tag{3.41}$$

obtained by partial integration. Certainly, this was expected because of the unsymmetric operator. Using equation (3.41), the extra boundary integral would require a seperate loop over boundary edges, adding (unsymmetrically) only to node j.

Observe that we take a difference on the edge level and then add contributions to both the end points. This implies that the conservation law given for the first derivatives is not reflected at the edge level, although it is still maintained at the point level. For a second form reflecting the conservation property on the edge level please refer to Löhner (2001).

3.3.5. Consistent and Lumped Mass Matrix

For the shape-functions themselves, a conservation property similar to equation (3.30) exists

$$N_i = 1 - \sum_{j \neq i} N_j. \tag{3.42}$$

Nevertheless, it is completely worthless if we attempt to use it with the objective of simplifying the implementation of mass terms. Due to the fact that we do not consider the diagonal entries of the element matrices in the build-up process, the missing term m_{ii} in fact is a problem for us.

To circumvent this inconvenience we decided to store the lumped mass matrix in the form of a nodal parameter list `mLumpedMassMatrix`. This makes it possible to use either the lumped version directly, which facilitates the programming but may not always be adequate, or to calculate the missing diagonal elements "on the fly" by summing up edge-contributions in a temporary variable and making use of

$$m_{ii} = m_i^{lumped} - \sum_{j \neq i} m_{ij} \tag{3.43}$$

before switching over to the next row start index.

3.4. Implementation for Incompressible Flows

In the intent of developping a general approach for incompressible and compressible flows, the low Mach number setting is a critical situation for the compressible case. As the Mach number approaches zero, compressible flow solvers suffer severe deficiencies, both in efficiency and accuracy. Principally there are two main approaches:

1. the **modification of a compressible solver** (density-based) downward to low Mach numbers or
2. the **extension of an incompressible solver** (pressure-based) towards this regime.

As already mentioned, this work focuses on the subsonic regime when the magnitude of the flow velocity is small compared with the acoustic wave-speed. In this case the dominance of the convection terms within the time-dependent equations renders the system stiff and causes density-based solvers to converge slowly. Time-marching procedures may suffer severe stability and accuracy restrictions so that they become inefficient for low Mach number flows. To capture solution convergence for these regimes two techniques – preconditioning and asymptotic schemes – have been proposed and are detailled in Keshtiban et al. (2004).

In contrast, pressure-based methods were originally conceived to solve incompressible flows, adopting pressure as a primary variable. Following this approach, pressure variation remains finite – irrespective of the Mach number –, which renders the computation tractable throughout the entire spectrum of Mach numbers. This has been the main reason for electing this path.

3.4.1. Problem Statement

Let Ω be the domain of $\mathbb{R}^n$ occupied by the fluid, where $n = 2$ or 3 is the number of space dimensions, $\Gamma = \partial\Omega$ its boundary and $[0, T]$ the time interval of analysis. The Euler problem (Table 3.2) consists in finding a velocity $\boldsymbol{u}$ and a kinematic pressure $p_{kin} = p/\rho$ such that

$$\begin{aligned}
\frac{\partial \boldsymbol{u}}{\partial t} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\boldsymbol{u} + \boldsymbol{\nabla} p_{kin} &= \boldsymbol{f} && \text{in } \Omega,\ t \in [0,T] \\
\boldsymbol{\nabla} \cdot \boldsymbol{u} &= 0 && \text{in } \Omega,\ t \in [0,T] \\
\boldsymbol{u} &= \boldsymbol{u}_D && \text{on } \Gamma_D,\ t \in [0,T] \\
p &= p_\infty && \text{on } \Gamma_N,\ t \in [0,T] \\
\boldsymbol{u} &= \boldsymbol{u}_0 && \text{in } \Omega,\ t = 0
\end{aligned} \tag{3.44}$$

where $\boldsymbol{f}$ is the force vector (per unit mass as well) and $\boldsymbol{u}_0$ represents the initial velocity field. The Dirichlet boundary condition states $\boldsymbol{u}_D = \boldsymbol{0}$ in the case of a no-slip boundary and $\boldsymbol{u}_D = \boldsymbol{u} - \boldsymbol{u} \cdot \boldsymbol{n}$ if a slip condition is applied, $\boldsymbol{n}$ being the outward unit normal. Γ_D and the Neumann boundary Γ_N, on which the external pressuer p_∞ is given, are disjoint components of Γ.

To write the weak form of the problem (3.44) we need to introduce some notation (Codina and Folch, 2004). We denote by $\mathcal{H}^1(\Omega)$ the Sobolev space of functions whose first derivatives belong to $\mathcal{L}^2(\Omega)$, and by $\mathcal{H}^1_0(\Omega)$ the subspace of $\mathcal{H}^1(\Omega)$ of functions with zero trace on Γ. A bold character is used for the vector counterpart of these spaces. The $\mathcal{L}^2(\Omega)$ scalar product in a set ω is denoted by $(\cdot,\cdot)_\omega$. The subscript ω is omitted when it coincides with Ω. To pose the problem, we also need the functional spaces $\boldsymbol{\mathcal{V}}_{st} = \boldsymbol{\mathcal{H}}^1_0(\Omega)^n$ and $\mathcal{Q}_{st} = \{q \in \mathcal{L}^2(\Omega) | \int_\Omega q = 0\}$ as well as $\boldsymbol{\mathcal{V}} = \boldsymbol{\mathcal{L}}^2(0,T;\boldsymbol{\mathcal{V}}_{st})$ and $\mathcal{Q} = \mathcal{L}^2(0,T;\mathcal{Q}_{st})$ for the transient problem.
Assuming for simplicity the force vector to be square integrable, the weak form of (3.44) consists in finding $(\boldsymbol{u}, p_{kin}) \in \boldsymbol{\mathcal{V}} \times \mathcal{Q}$ such that

$$(\partial_t \boldsymbol{u}, \boldsymbol{\nu}) + (\boldsymbol{u} \cdot \boldsymbol{\nabla}\boldsymbol{u}, \boldsymbol{\nu}) - (p_{kin}, \boldsymbol{\nabla} \cdot \boldsymbol{\nu}) = (\boldsymbol{f}, \boldsymbol{\nu}) \qquad \forall \boldsymbol{\nu} \in \boldsymbol{\mathcal{V}}_{st}, \tag{3.45a}$$

$$(q, \boldsymbol{\nabla} \cdot \boldsymbol{u}) = 0 \qquad \forall q \in \mathcal{Q}_{st}, \tag{3.45b}$$

and satisfying the initial condition in a weak sense.

3.4.2. Discretization

Principally, any **temporal discretization** is possible. However, we shall concentrate on the monolithic (solving for velocity and pressure at the same time) backward Euler scheme. The time discretized version of (3.45) requires, from known $\boldsymbol{u}^n$, to find $\boldsymbol{u}^{n+1} \in \boldsymbol{\mathcal{V}}_h$ and $p^{n+1}_{kin} \in \mathcal{Q}_h$ such that

$$(\delta_t \boldsymbol{u}^n, \boldsymbol{\nu}) + (\boldsymbol{u}^{n+1} \cdot \boldsymbol{\nabla}\boldsymbol{u}^{n+1}, \boldsymbol{\nu}) - (p^{n+1}_{kin}, \boldsymbol{\nabla} \cdot \boldsymbol{\nu}) = (\bar{\boldsymbol{f}}^{n+1}, \boldsymbol{\nu}) \qquad \forall \boldsymbol{\nu} \in \boldsymbol{\mathcal{V}}_{st}, \tag{3.46a}$$

$$(q, \boldsymbol{\nabla} \cdot \boldsymbol{u}^{n+1}) = 0 \qquad \forall q \in \mathcal{Q}_{st}. \tag{3.46b}$$

The notation $\delta_t \boldsymbol{u}^n := \frac{\Delta \boldsymbol{u}^n}{\Delta t}$ and $\Delta \boldsymbol{u}^n = \boldsymbol{u}^{n+1} - \boldsymbol{u}^n$ has been used. The term $\bar{\boldsymbol{f}}^{n+1}$ has to be understood as the time average of the force in the interval $[t^n, t^{n+1}]$. The time step size $\Delta t = t^{n+1} - t^n$ is computed using the *Courant-Friedrichs-Lewy* (CFL) *condition*, which is commonly (Wikipedia – The Free Encyclopedia) represented as

$$\frac{u \Delta t}{\Delta x} < C \tag{3.47}$$

for one-dimensional pure advection (ignoring diffusion or reaction terms) schemes. u is the velocity, Δt the time step, Δx the length interval and C a constant depending on the particular equation to be solved and not on Δt and Δx. Using an edge-based data structure we will introduce the nodal parameter

$$h_{i,min} = \min_j l_{ij} \tag{3.48}$$

as the minimum length l_{ij} of the edges ij surrounding the node i in order to approximate Δx. Hence the time step is computed as

$$\Delta t = \min_i \left(\frac{h_{i,min}}{|\boldsymbol{u}_i|} \right) N_{CFL} \tag{3.49}$$

where the *Courant number* N_{CFL} shall incorporate a certain security factor. It can be defined in the Python start script of the simulation.

With regard to **spatial discretization**, let $\boldsymbol{\mathcal{V}}_h$ be a finite element space to approximate $\boldsymbol{\mathcal{V}}$, and $\mathcal{Q}_h$ a finite element approximation to $\mathcal{Q}$. Functions in $\boldsymbol{\mathcal{V}}_h$ need to be continuous piecewise polynomials, whereas continuity principally is not necessary for $\mathcal{Q}_h$. However, for reasons explained below, we will consider only continuous pressure interpolations. It is well known that for this discrete problem to be stable

$$(\delta_t \boldsymbol{u}_h^n, \boldsymbol{\nu}_h) + (\boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} \boldsymbol{u}_h^{n+1}, \boldsymbol{\nu}_h) - (p_{kin,h}^{n+1}, \boldsymbol{\nabla} \cdot \boldsymbol{\nu}_h) = (\bar{\boldsymbol{f}}^{n+1}, \boldsymbol{\nu}_h) \quad \forall \boldsymbol{\nu}_h \in \boldsymbol{\mathcal{V}}_h, \tag{3.50a}$$

$$(q_h, \boldsymbol{\nabla} \cdot \boldsymbol{u}_h^{n+1}) = 0 \quad \forall q_h \in \mathcal{Q}_h. \tag{3.50b}$$

the velocity and pressure spaces need to satisfy the classical inf-sup condition, which in particular precludes the use of convenient *equal* velocity-pressure interpolations. However, it can be demonstrated that this condition is not required when fractional step methods using a pressure Poisson equation are employed (Codina, 2001).

Before introducing the fractional step scheme, the matrix form of the problem shall be presented

$$\mathsf{M}\frac{\mathsf{u}^{n+1} - \mathsf{u}^n}{\Delta t} + \mathsf{C}(\mathsf{u}^{n+1})\mathsf{u}^{n+1} + \mathsf{G}\mathsf{p}^{n+1} = \mathsf{f}^{n+1} \tag{3.51a}$$

$$\mathsf{D}\mathsf{u}^{n+1} = 0 \tag{3.51b}$$

where u and p are the arrays of nodal velocities and kinematic pressures, respectively. Keeping the index conventions introduced in Section 3.2.2 (indices k and l to indicate space dimensions, whereas i and j are employed for nodal values and shape functions), the components of the arrays envolved in the discrete problem (3.51) are

$$\begin{aligned}
\mathsf{M}_{ij,kl} &= (N_i, N_j)\delta_{kl}, \\
\mathsf{C}(\mathsf{u}^{n+1})_{ij,kl} &= (N_i, \boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} N_j)\delta_{kl}, \\
\mathsf{G}_{ij,k} &= (N_i, \partial_k N_j) = -(\partial_k N_i, N_j), \\
\mathsf{f}_{i,k} &= (N_i, f_k), \\
\mathsf{D}_{ij,l} &= (N_i, \partial_l N_j),
\end{aligned} \tag{3.52}$$

where δ_{kl} is the Kronecker Delta. Note the property $\mathsf{G} = -\mathsf{D}^T$. Except f, which is a vector, all the arrays are matrices whose components can be obtained by grouping together first spatial and nodal index (k and possibly i) and doing the same for the second indices (l and possibly j). Though, we do not really "construct" these matrices as we will use the precalculated edge-data presented in Section 3.3 to perform the operations.

In this context the contributions of the convective Galerkin term have to be mentioned. In an element-based implementation they are computed as

$$\mathsf{C}_{ij,ll} = \sum_{k=1}^{n_{dof}} \int_\Omega N_i a_k \frac{\partial N_j}{\partial x_k} \mathrm{d}\Omega, \tag{3.53}$$

where a_k is the k-th component of the advective velocity ($\boldsymbol{a} = \boldsymbol{u}_h^{n+1}$ in (3.50)). In order to use the pre-computed gradient matrix $\mathbf{G}$, the following approximation must be done at this point according to Soto et al. (2004):

$$\mathsf{C}_{ij,ll} \approx \sum_{k=1}^{n_{dof}} a_{ij,k} \int_\Omega N_i \frac{\partial N_j}{\partial x_k} \mathrm{d}\Omega, \tag{3.54}$$

where $a_{ij,k}$ is the k-th component of the advective velocity associated with the edge ij. The first idea is to take $\boldsymbol{a}_{ij}$ as the average velocity of the nodal points i and j and to enforce the conservation property $\sum_{j=1}^{n_{pts}} \left(\sum_{k=1}^{n_{dof}} \mathsf{C}_{ij,lk} \right) = 0$ by computing the diagonal as the subtraction of the same-row non-diagonal terms. However, this procedure would destroy the second-order Galerkin, or central-difference, approximation of the convective term (assuming that linear elements are used). Such a second-order approximation is reflected at discrete level by the fact that $\mathsf{C}_{ii,ll} = 0$ for the interior nodal points, something that naturally arises in a standard finite element approximation using linear elements (or in finite differences using a central scheme).
The only way to fulfill this condition and to maintain the consistency of the method – the exact solution is still a solution of the discrete problem – is taking $\boldsymbol{a}_{ij}$ as a function of only the nodal point i ($\boldsymbol{a}_{ij} = \boldsymbol{a}_i$). In this case it is easy to verify for the interior points that

$$\mathsf{C}_{ii,ll} \approx \sum_{jk \neq il} a_{i,k} \int_\Omega N_i \frac{\partial N_j}{\partial x_k} \mathrm{d}\Omega = 0, \tag{3.55}$$

implying that the approximation is of second-order and the conservation property for stationary terms holds.

Concerning the computation of $\boldsymbol{a}_i$, the velocity at the nodal point i, $\boldsymbol{u}_i$, certainly is first choice. Nevertheless, we used the following smoothing in this work

$$a_{i,k} = \frac{m_{ij} u_{j,k}}{\sum_{j \neq i} m_{ij}} \qquad \forall j \neq i \qquad \text{for interior points} \tag{3.56a}$$

$$a_{i,k} = \frac{m_{ij} u_{j,k}}{\sum_{j=1}^{n_{pts}} m_{ij}} \qquad \forall j \qquad \text{for boundary points} \tag{3.56b}$$

where m_{ij} is the term ij of the consistent mass matrix and $u_{j,k}$ is the k-th component of the velocity at nodal point j. Note that the computation is only done over the neighbours j connected to point i – using the explained CSR-loop over edges – as otherwise the contribution is zero.
Numerical experience indicates that equations (3.56) for the advective velocity associated with the nodal point i produce a better convergence rate and more accurate results then taking simply $\boldsymbol{a}_i = \boldsymbol{u}_i$ (Soto et al., 2004). Moreover, it can be checked that, given a discrete velocity field $\boldsymbol{u}$, the convective RHS obtained employing a standard element-based implementation

$$f_{i,l}^{element} = \sum_{j=1}^{n_{pts}} \left(\sum_{k=1}^{n_{dof}} \int_\Omega N_i u_k \frac{\partial N_j}{\partial x_k} \mathrm{d}\Omega u_{j,l} \right) \tag{3.57}$$

is much better approximated by using equations (3.56). Besides, they naturally arise from a central difference (or central finite volume) discretization of the convective Navier-Stokes term.

Note that the same treatment will be applicable to the convective stabilization terms introduced in Section 3.4.4.

3.4.3. Fractional Step Algorithm

The fully discrete problem (3.51) is exactly equivalent to

$$\mathbf{M}\frac{1}{\Delta t}(\tilde{\mathsf{u}}^{n+1}-\mathsf{u}^{n})+\mathbf{C}(\mathsf{u}^{n+1})\mathsf{u}^{n+1}+\gamma\mathbf{G}\mathsf{p}^{n}=\mathsf{f}^{n+1} \tag{3.58a}$$

$$\mathbf{M}\frac{1}{\Delta t}(\mathsf{u}^{n+1}-\tilde{\mathsf{u}}^{n+1})+\mathbf{G}(\mathsf{p}^{n+1}-\gamma\mathsf{p}^{n})=0 \tag{3.58b}$$

$$\mathbf{D}\mathsf{u}^{n+1}=0 \tag{3.58c}$$

as the splitting of the momentum equation is purely algebraic. The array $\tilde{\mathsf{u}}^{n+1}$ contains nodal values of the auxiliary variable called *fractional velocity*, and γ is a numerical parameter whose values of interest are 0 (first-order splitting) and 1 (second-order splitting). At this point we can make the essential approximation

$$\mathbf{C}(\mathsf{u}^{n+1})\mathsf{u}^{n+1}\approx\mathbf{C}(\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1} \tag{3.59}$$

which may be interpreted as an incomplete block LU factorization of the original problem (3.58). The advantage of this discrete approach is that now there is no question about the boundary conditions for the intermediate variable $\tilde{\mathsf{u}}^{n+1}$: since boundary conditions are incorporated in the discrete problem (3.58), the prescriptions for the fractional velocity are exactly the same as for the end-of-step velocity u^{n+1} (Codina, 2001).
By means of equation (3.58b), u^{n+1} can be expressed in terms of $\tilde{\mathsf{u}}^{n+1}$ and inserted into equation (3.58c), which yields the following set of equations to be solved

$$\mathbf{M}\frac{1}{\Delta t}(\tilde{\mathsf{u}}^{n+1}-\mathsf{u}^{n})+\mathbf{C}(\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1}+\gamma\mathbf{G}\mathsf{p}^{n}=\mathsf{f}^{n+1} \tag{3.60a}$$

$$\mathbf{M}\frac{1}{\Delta t}(\mathsf{u}^{n+1}-\tilde{\mathsf{u}}^{n+1})+\mathbf{G}(\mathsf{p}^{n+1}-\gamma\mathsf{p}^{n})=0 \tag{3.60b}$$

$$\Delta t\mathbf{D}\mathbf{M}^{-1}\mathbf{G}(\mathsf{p}^{n+1}-\gamma\mathsf{p}^{n})=\mathbf{D}\tilde{\mathsf{u}}^{n+1} \tag{3.60c}$$

Even though the problem can be implemented as such, it is very convenient to make a further approximation. In order to avoid dealing with the matrix $\mathbf{D}\mathbf{M}^{-1}\mathbf{G}$ we shall approximate it by the Laplacian operator

$$\mathbf{D}\mathbf{M}^{-1}\mathbf{G}\approx\mathbf{L},\quad\text{with components}\quad \mathsf{L}_{ij}=-(\boldsymbol{\nabla}N_i,\boldsymbol{\nabla}N_j). \tag{3.61}$$

It shall be stated here that this approximation is only possible when continuous pressure interpolations are employed. Likewise, it introduces implicitly the same wrong pressure boundary condition as when the splitting is performed at continuous level (see Codina (2001) for a detailed discussion).

After having ordered according to the sequence of solution (first $\tilde{\mathsf{u}}^{n+1}$, then p^{n+1} and finally u^{n+1}) the problem to be solved is

$$\mathbf{M}\frac{1}{\Delta t}(\tilde{\mathsf{u}}^{n+1} - \mathsf{u}^n) + \mathbf{C}(\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1} + \gamma\mathbf{G}\mathsf{p}^n = \mathsf{f}^{n+1} \tag{3.62a}$$

$$\Delta t\mathbf{L}(\mathsf{p}^{n+1} - \gamma\mathsf{p}^n) = \mathbf{D}\tilde{\mathsf{u}}^{n+1} \tag{3.62b}$$

$$\mathbf{M}\frac{1}{\Delta t}(\mathsf{u}^{n+1} - \tilde{\mathsf{u}}^{n+1}) + \mathbf{G}(\mathsf{p}^{n+1} - \gamma\mathsf{p}^n) = 0. \tag{3.62c}$$

3.4.4. Stabilization Techniques

Pressure and Convection Stabilization

The treatment of **pressure** in numerical approximations of incompressible flow problems is still an active subject of research, basically for two reasons: On the one hand its approximation needs to be different from that of the velocity field in order to obtain a stable numerical scheme. On the other hand its coupling with the velocity components makes the solution of the linear system, arising from the discretization of the equations, highly demanding from a computational point of view (Codina and Soto, 2004).

- Referring to the *pressure approximation*, the use of finite element methods leads to the well known inf-sup stability condition for the velocity and pressure finite element spaces if the standard Galerkin formulation is used. Either one uses velocity pressure pairs fulfilling the inf-sup condition, or the discrete variational formulation of the problem has to be modified in order to circumvent it. Finite element formulations of this kind may fall basically into two categories: techniques to stabilize simple elements, such as the Q_1/P_0 pair (multilinear velocity, piecewise constant pressure) and methods that allow the use of *equal* interpolations (and therefore continuous pressures). We will apply the latter in this work.
- Concerning the *velocity-pressure coupling*, fractional step methods for the incompressible Navier-Stokes equations enjoy widespread popularity because of their computational efficiency based on the uncoupling of of the pressure from the velocity components. However, several issues related to these methods still deserve further analysis, and perhaps the computed pressure near boundaries and the stability of the pressure itself.

Apart from the pressure treatment, another important issue to be considered in the numerical approximation of incompressible flows is the numerical instability problem, found when the viscous term is small compared to the **convective** one – which is evident in the Euler problem ($\nu = 0$). Both, the inf-sup condition as well as the convection instabilities, can be overcome by resorting from the standard Galerkin method to a *stabilized* formulation. The one adopted in this work is based on the subgrid scale concept. The basic idea is to approximate the effect of the component of the continuous solution that cannot be resolved by the finite element mesh on the discrete finite element solution. Hence an important feature of the formulation is that the unresolved component, hereafter referred to as *subgride scale* or *subscale*, is assumed to be $\mathcal{L}^2$ orthogonal to the finite element space.

Orthogonal Subscale Stabilization

Starting once more with the weak form of the problem, the discrete problem is obtained by approximating $\boldsymbol{u}$ and p_{kin}. If $\boldsymbol{u}_h$ and $p_{kin,h}$ are the finite element unknowns, we put $\boldsymbol{u} \approx \boldsymbol{u}_h + \hat{\boldsymbol{u}}$ and $p_{kin} \approx p_{kin,h}$. That is to say that the velocity is approximated by its finite element component plus an additional term whereas the pressure subscale will be taken as zero for the sake of simplicity. $\boldsymbol{u}^n \approx \boldsymbol{u}_*^n := \boldsymbol{u}_h^n + \hat{\boldsymbol{u}}^n$ and $p_{kin}^n \approx p_{kin,h}^n$ are called the velocity and the (kinematic) pressure for time level n. Considering the spatial discretization, we assume that $\boldsymbol{u}_h^n$ and $p_{kin,h}^n$ are constructed using the standard finite element interpolation. In particular, equal velocity-pressure interpolation is possible with the orthogonal subscale stabilization. Concerning the behaviour of $\hat{\boldsymbol{u}}^n$, a bubble-like function is assumed so that it vanishes on the interelement boundaries. However, contrary to what is commonly done, no particular behaviour of the velocity subscale is assumed within the element domains. Following closely the operations and modifications outlined in Codina and Soto (2004), one finally arrives at the discrete problem

$$\begin{aligned}(\delta_t \boldsymbol{u}_h^n, \boldsymbol{\nu}_h) + (\boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} \boldsymbol{u}_h^{n+1}, \boldsymbol{\nu}_h) - (p_{kin,h}^{n+1}, \boldsymbol{\nabla} \cdot \boldsymbol{\nu}_h) \\ + (\tau P_h^{\perp}(\boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} \boldsymbol{u}_h^{n+1}), \boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} \boldsymbol{\nu}_h) = (\bar{\boldsymbol{f}}^{n+1}, \boldsymbol{\nu}_h) \qquad \forall \boldsymbol{\nu}_h \in \boldsymbol{\mathcal{V}}_h, \qquad (3.63a)\\ (q_h, \boldsymbol{\nabla} \cdot \boldsymbol{u}_h^{n+1}) + (\tau P_h^{\perp}(\boldsymbol{\nabla} p_{kin,h}^{n+1}), \boldsymbol{\nabla} q_h) = 0 \qquad \forall q_h \in \mathcal{Q}_h. \qquad (3.63b)\end{aligned}$$

where the *orthogonal projections* can be expressed as $P_h^{\perp} = I - P_h$ with P_h being the $\mathcal{L}^2$-projection onto $\boldsymbol{\mathcal{V}}_h$:

$$P_h^{\perp}(\boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} \boldsymbol{u}_h^{n+1}) = \boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} \boldsymbol{u}_h^{n+1} - \boldsymbol{y}_h^{n+1}, \qquad (3.64a)$$

$$P_h^{\perp}(\boldsymbol{\nabla} p_h^{n+1}) = \boldsymbol{\nabla} p_h^{n+1} - \boldsymbol{z}_h^{n+1}. \qquad (3.64b)$$

$\boldsymbol{y}_h^{n+1}$ and $\boldsymbol{z}_h^{n+1}$ are the solution of

$$(\boldsymbol{y}_h^{n+1}, \boldsymbol{\nu}_h) = (\boldsymbol{u}_h^{n+1} \cdot \boldsymbol{\nabla} \boldsymbol{u}_h^{n+1}, \boldsymbol{\nu}_h) \qquad \forall \boldsymbol{\nu}_h \in \boldsymbol{\mathcal{V}}_h, \qquad (3.65a)$$

$$(\boldsymbol{z}_h^{n+1}, \boldsymbol{\nu}_h) = (\boldsymbol{\nabla} p_{kin,h}^{n+1}, \boldsymbol{\nu}_h) \qquad \forall \boldsymbol{\nu}_h \in \boldsymbol{\mathcal{V}}_h. \qquad (3.65b)$$

The stability and convergence analysis for the Navier-Stokes problem dictates that the *intrinsic time* τ, like it is called by Soto et al. (2004), must be computed as

$$\tau = \frac{h^2}{4\nu + 2|\boldsymbol{u}|h} \qquad (3.66)$$

where h and $\boldsymbol{u}$ are the typical element size and velocity respectively, and ν is the viscosity of the fluid. Particularly with regard to our edge-based data structure and the envisaged nodal implementation, we use once more the minimum edge-length $h_{i,min}$ calculated by equation (3.48). The "inviscid version" of equation (3.66) states

$$\tau_i = \frac{h_i}{2|\boldsymbol{u}_i| + \varepsilon}. \qquad (3.67)$$

By the choice of a relatively small parameter ε we guarantee that the denominator is different from zero. In the case of compressible Euler equations the influence of "reaction

terms" may be taken into account, which would render this last measure unnecessary. Note that τ has been included within the inner product since, in principle, it changes from point to point. The terms multiplied by this parameter are responsible for the enhancement of stability with respect to the standrad Galerkin method, which is why we call them *stabilization terms*.

Considering these in the matrix form of the fractional step scheme yields

$$\begin{aligned}
&\mathbf{M}\frac{1}{\Delta t}(\tilde{\mathsf{u}}^{n+1}-\mathsf{u}^{n})+\mathbf{C}(\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1}+\gamma\mathbf{G}\mathsf{p}^{n}\\
&\qquad+\mathbf{S}_u(\tilde{\tau}^{n+1};\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1}-\mathbf{S}_y(\tilde{\tau}^{n+1};\tilde{\mathsf{u}}^{n+1})\mathsf{y}^{n+1}=\mathsf{f}^{n+1} &\text{(3.68a)}\\
&\Delta t\mathbf{L}(\mathsf{p}^{n+1}-\gamma\mathsf{p}^{n})+\mathbf{S}_p(\tilde{\tau}^{n+1})\mathsf{p}^{n+1}-\mathbf{S}_z(\tilde{\tau}^{n+1})\mathsf{z}^{n+1}=\mathbf{D}\tilde{\mathsf{u}}^{n+1} &\text{(3.68b)}\\
&\mathbf{M}\frac{1}{\Delta t}(\mathsf{u}^{n+1}-\tilde{\mathsf{u}}^{n+1})+\mathbf{G}(\mathsf{p}^{n+1}-\gamma\mathsf{p}^{n})=0 &\text{(3.68c)}\\
&\mathbf{M}\mathsf{y}^{n+1}-\mathbf{C}(\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1}=0 &\text{(3.68d)}\\
&\mathbf{M}\mathsf{z}^{n+1}-\mathbf{G}\mathsf{p}^{n+1}=0. &\text{(3.68e)}
\end{aligned}$$

where the components of the stabilization arrays are

$$\begin{aligned}
\mathsf{S}_u(\tilde{\tau}^{n+1};\tilde{\mathsf{u}}^{n+1})_{ij,kl}&=(\tilde{\tau}_i^{n+1}\tilde{\boldsymbol{u}}_h^{n+1}\cdot\boldsymbol{\nabla}N_i,\tilde{\boldsymbol{u}}_h^{n+1}\cdot\boldsymbol{\nabla}N_j)\delta_{kl}\\
\mathsf{S}_y(\tilde{\tau}^{n+1};\tilde{\mathsf{u}}^{n+1})_{ij,kl}&=(\tilde{\tau}_i^{n+1}\tilde{\boldsymbol{u}}_h^{n+1}\cdot\boldsymbol{\nabla}N_i,N_j)\delta_{kl}\\
\mathsf{S}_p(\tilde{\tau}^{n+1})_{ij}&=(\tilde{\tau}_i^{n+1}\boldsymbol{\nabla}N_i,\boldsymbol{\nabla}N_j)\\
\mathsf{S}_z(\tilde{\tau}^{n+1})_{ij,l}&=(\tilde{\tau}_i^{n+1}\partial_l N_i,N_j)
\end{aligned}\qquad(3.69)$$

using the known convention for nodal indices i, j and spatial indices k, l. To avoid the resolution of an equation system for the projection terms, the lumped mass matrix is used so that equations (3.68d) and (3.68e) can be considered in an edge-based manner within (3.68a) and (3.68b) respectively:

$$\begin{aligned}
&\mathbf{M}\frac{1}{\Delta t}(\tilde{\mathsf{u}}^{n+1}-\mathsf{u}^{n})+\mathbf{C}(\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1}+\gamma\mathbf{G}\mathsf{p}^{n}\\
&\qquad+\mathbf{S}_u(\tilde{\tau}^{n+1};\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1}-\mathbf{S}_y(\tilde{\tau}^{n+1};\tilde{\mathsf{u}}^{n+1})\mathbf{M}^{-1}\mathbf{C}(\tilde{\mathsf{u}}^{n+1})\tilde{\mathsf{u}}^{n+1}=\mathsf{f}^{n+1} &\text{(3.70a)}\\
&\Delta t\mathbf{L}(\mathsf{p}^{n+1}-\gamma\mathsf{p}^{n})+\mathbf{S}_p(\tilde{\tau}^{n+1})\mathsf{p}^{n+1}-\mathbf{S}_z(\tilde{\tau}^{n+1})\mathbf{M}^{-1}\mathbf{G}\mathsf{p}^{n+1}=\mathbf{D}\tilde{\mathsf{u}}^{n+1} &\text{(3.70b)}\\
&\mathbf{M}\frac{1}{\Delta t}(\mathsf{u}^{n+1}-\tilde{\mathsf{u}}^{n+1})+\mathbf{G}(\mathsf{p}^{n+1}-\gamma\mathsf{p}^{n})=0 &\text{(3.70c)}
\end{aligned}$$

3.4.5. Solving Procedure and Boundary Conditions

Table 3.4 gives an overview of the implemented algorithm for the simulation of incompressible flows and details each step.

1. Load Kratos kernel and application modules
2. Read model part defined by GiD → Section 2.3.1
   - Change initial values and/or set ramp-up
   - Change boundary conditions → Section 2.2.3
3. Define simulation parameters
   - Set time interval and CFL number
   - Set solver tolerance and stop criteria
4. Create matrix container
   - Construct CSR vector → Listing B.1
   - Compute and store edge-based data → Listing B.2
5. Initialize flow solver
   - Set free-flow conditions
   - Choose linear solver for pressure Poisson equation
6. Loop over time steps and perform calculations
   a) Compute time step size
   b) Step 1: compute fractional velocity $\tilde{\mathsf{u}}^{n+1}$
      - Solve equation (3.70a)
      - Use Picard iteration to deal with non-linear terms
   c) Step 2: compute pressure variation $\Delta\mathsf{p}^{n}$ or directly p^{n+1}
      - Solve linear equation system (3.70b)
      - Penalize matrix entries for Neumann pressure nodes
   d) Step 3: compute end-of-step velocity u^{n+1}
      - Solve equation (3.70c)
      - Apply Dirichlet conditions for velocity (inlet, slip, no-slip)
   e) Output
      - Calculate dimensionless coefficients for post-processing
      - Write nodal results

Table 3.4: Solving procedure for incompressible flows

3.5. Expansion for Compressible Flows

3.5.1. Modifications

Let us recall the continuous **momentum equation** for the Euler problem first:

$$\frac{\partial(\rho u_l)}{\partial t} + \frac{\partial(\rho u_l u_k)}{\partial x_k} + \frac{\partial p}{\partial x_l} = \rho g_l \tag{3.71}$$

and introduce the momentum $U_l = \rho u_l$ as variable

$$\frac{\partial U_l}{\partial t} + \frac{\partial(U_l u_k)}{\partial x_k} + \frac{\partial p}{\partial x_l} = F_l. \tag{3.72}$$

Note that we are dealing with the real thermodynamic pressure p now and the force vector is defined by $\boldsymbol{F} = \rho \boldsymbol{g}$ this time. It is obvious that we obtain a slightly different convective term compared to the incompressible case. That is why the convective matrix $\mathbf{C}^*$ of the discrete problem is marked with an asterisk in the following. It is advisible to implement the term as a whole using the edge-based techniques for Euler fluxes mentioned in Section 3.3.4. Nevertheless, a linearized form $U_l \frac{\partial u_k}{\partial x_k} + u_k \frac{\partial U_l}{\partial x_k}$ may be used as well.

With regard to the conservation of mass, a new term appears in the **continuity equation**, namely the temporal derivative of the density

$$\frac{\partial \rho}{\partial t} + \frac{\partial U_k}{\partial x_k} = 0 \tag{3.73}$$

In the case of incompressible flows the continuity equation is formulated in terms of pressure only. Now, however, we have the possibility of choosing either the density or the pressure as unknown of the problem. As our aim is a general approach for incompressible and compressible flows we shall keep the pressure as variable. For formulations using the density as variable, please refer to Vázquez et al. (1999).
In order to replace the density variation by the pressure variation we make use of the relation

$$\Delta \rho^n = \alpha \Delta p^n \tag{3.74}$$

where α is a function that will be defined according to the type of flow being analyzed. We will see that it is useful to introduce the matrix $\mathbf{M}_\alpha$, of components

$$\mathsf{M}_{\alpha,ij} = \int_\Omega \alpha N_i N_j \mathrm{d}\Omega, \tag{3.75}$$

where N_i is the shape function associated to the i-th node of the finite element mesh with which we assume that all the variables are interpolated.

3.5.2. Generalization of the Algorithm

At this point we make use of the simplifications mentioned in Section 3.2.1 that allow us the simulation of compressible flows in subsonic regime without resolving the energy equation. The following types of flow are distinguished in this context:

Incompressible flows are characterized by the equations in Table 3.2. Hence it is evident that the time variation of the density has to disappear, which directly demands $\alpha = 0$.

Slightly compressible flows are approximated using equation (3.16) with a constant value for the speed of sound, so that $\alpha = \frac{1}{c^2}$ is chosen.

Barotropic flows only involve density and pressure in the equation of state. Resorting to relation (3.18), we use $\alpha = \frac{\rho^n}{\gamma p^n}$ where the superscript n indicates the time step. Accordingly, the nodal function value α is calculated at the end of the previous time step.

Perfect gases are not covered in this work as they require the solution of the energy equation. In this case the equation of state is used to link pressure and density, resulting in the choice $\alpha = \frac{1}{RT}$. Furthermore the RHS of the continuity equation has to be modified due to the variation in time of the temperature. For detailed information please refer to Vázquez et al. (1999).

$$\alpha = \begin{cases} 0 & \text{for incompressible flows} \\ \frac{1}{c^2} & \text{for slightly compressible flows} \\ \frac{\rho^n}{\gamma p^n} & \text{for barotropic flows (isentropic perfect gases)} \end{cases} \tag{3.76}$$

3.5.3. Modified Fractional Step Scheme

Taking into account the mentioned modifications in the equation system (3.62), the fractional step algorithm for compressible flows can be written in the following form:

$$\mathbf{M}\frac{1}{\Delta t}(\tilde{\mathsf{U}}^{n+1} - \mathsf{U}^n) + \mathbf{C}^*(\tilde{\mathsf{U}}^{n+1})\tilde{\mathsf{U}}^{n+1} + \gamma \mathbf{G}\mathsf{P}^n = \mathsf{F}^{n+1} \tag{3.77a}$$

$$\mathbf{M}_\alpha \frac{\Delta \mathsf{P}^n}{\Delta t} + \Delta t \mathbf{L}(\mathsf{P}^{n+1} - \gamma \mathsf{P}^n) = \mathbf{D}\tilde{\mathsf{u}}^{n+1} \tag{3.77b}$$

$$\mathbf{M}\frac{1}{\Delta t}(\mathsf{U}^{n+1} - \tilde{\mathsf{U}}^{n+1}) + \mathbf{G}(\mathsf{P}^{n+1} - \gamma \mathsf{P}^n) = 0. \tag{3.77c}$$

This time we resolve for the fractional momentum $\tilde{\mathsf{U}}^{n+1}$ first, followed by the pressure P^{n+1} and the end-of-step momentum U^{n+1}. Finally the density ρ^{n+1} is calculated by equation (3.74) and is used to extract the velocity u^{n+1} from the end-of-step momentum. Exactly the same stabilization terms as presented in Section 3.4.4 have been used, the fractional momentum replacing the fractional velocity in the compressible case. Just for the purpose of clarity they do not appear in the equations above.

3.5.4. General Solving Procedure

Table 3.5 gives an overview of the resulting general algorithm that might be used for incompressible flow simulations as well, provided that the freestream conditions are set accordingly.

1. Load Kratos kernel and application modules
2. Read model part defined by GiD → Section 2.3.1
   - Change initial values and/or set ramp-up
   - Change boundary conditions → Section 2.2.3
3. Define simulation parameters
   - Set time interval and CFL number
   - Set solver tolerance and stop criteria
4. Create matrix container
   - Construct CSR vector → Listing B.1
   - Compute and store edge-based data → Listing B.2
5. Initialize flow solver
   - Set free-flow conditions
   - Choose linear solver for pressure Poisson equation
6. Loop over time steps and perform calculations
   a) Compute time step size
   b) Step 1: compute fractional momentum $\tilde{\mathsf{U}}^{n+1}$
      - Solve equation (3.77a)
      - Use Picard iteration to deal with non-linear terms
   c) Step 2: compute pressure variation $\Delta\mathsf{P}^n$ or directly P^{n+1}
      - Solve linear equation system (3.77b)
      - Penalize matrix entries for Neumann pressure nodes
      - Compute density variation $\Delta\rho^n$ (3.74) and finally ρ^{n+1}
   d) Step 3: compute end-of-step momentum U^{n+1}
      - Solve equation (3.77c)
      - Extract velocity u^{n+1} using ρ^{n+1}
      - Apply Dirichlet conditions for velocity (inlet, slip, no-slip)
   e) Step 4: prepare next time step
      - Compute α using equation (3.76)
   f) Output
      - Calculate dimensionless coefficients for post-processing
      - Write nodal results

Table 3.5: Solving procedure for compressible and incompressible flows

3.6. Numerical Examples

Before preparing the flow solver for the fluid-structure coupling, the implementation shall be validated by numerical results in two and three dimensions. In a first step the focus will be on the verification of the edge-based data structure. Subsequently the implemented algorithm is going to be checked.

3.6.1. Cube with Quiescent Water

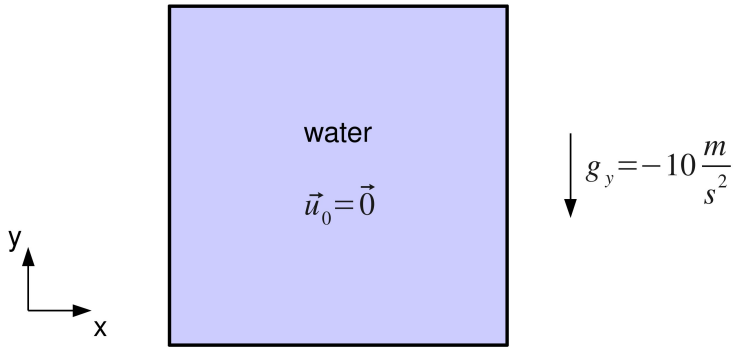


Figure 3.2: Cube with quiescent water – geometry and problem definition

As the velocity field has been initialized with zero and a no-slip condition is applied on the whole boundary, convective terms do not account for in the flow equations. The consideration of gravity ($g_y = -10\frac{m}{s^2}$) allows us to get a first impression of the pressure gradients calculated by the Poisson equation whose correct implementation and whose stability in the stationary regime are revealed by the pressure distribution in Figure 3.3.

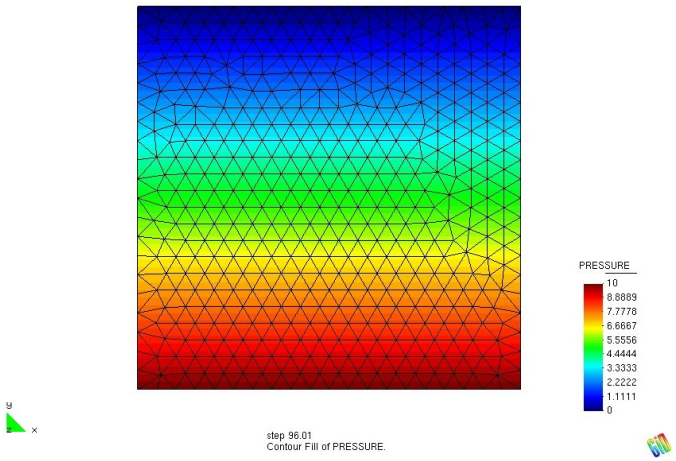


Figure 3.3: Pressure distribution of quiescent water under the influence of gravity

3.6.2. Airflow around a Cylinder

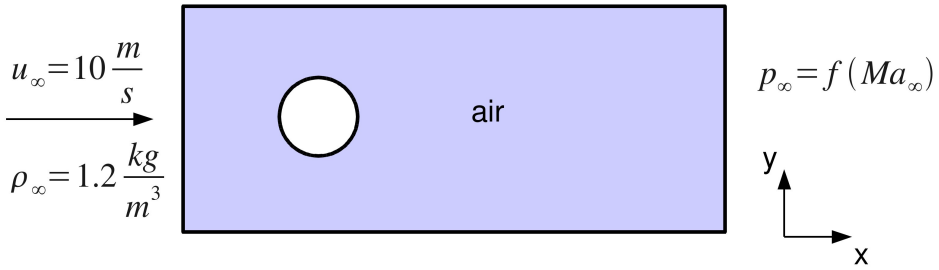


Figure 3.4: Airflow around a cylinder – geometry and problem definition

Having checked pressure and gravity in a quiescent situation, we will now set the fluid in motion. However, before doing so we will focus one last time on the edge-based data structure that we used for the implementation of the algorithm. In Section 3.3.4 the gradient and the transposed gradient were defined by equations (3.36) and (3.40). A comparison of g_{ij} and g_{ji} revealed that they are linked by a boundary integral due to partial integration (3.41)

$$g_{ij,k} + g_{ij,k} = \int_\Gamma N_j N_i n_k \mathrm{d}\Gamma. \tag{3.78}$$

Making use of this property and applying the sum of the two gradients to a constant pressure field of the value 1 has to give the following result:

- zero in the interior of the domain and
- the outward area normal of the faces on the boundary.

We used this test to validate the implementation of the two gradients, which is illustrated in Figure 3.5. Considering the dimensions of the domain, the nodal area normal was calculated by hand, which proved to be congruent with the result of operation (3.78).

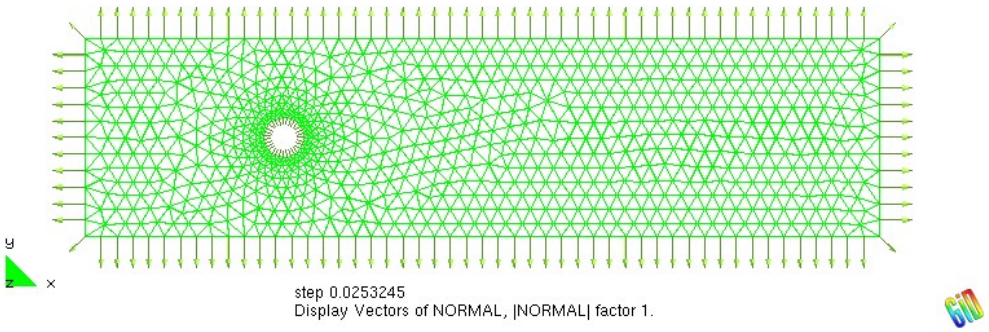


Figure 3.5: Validation of the gradient implementation

By the way, for further calculations area and unit normals have been computed using geometry data of the finite element mesh and *not* using the gradients.

Incompressible Flow

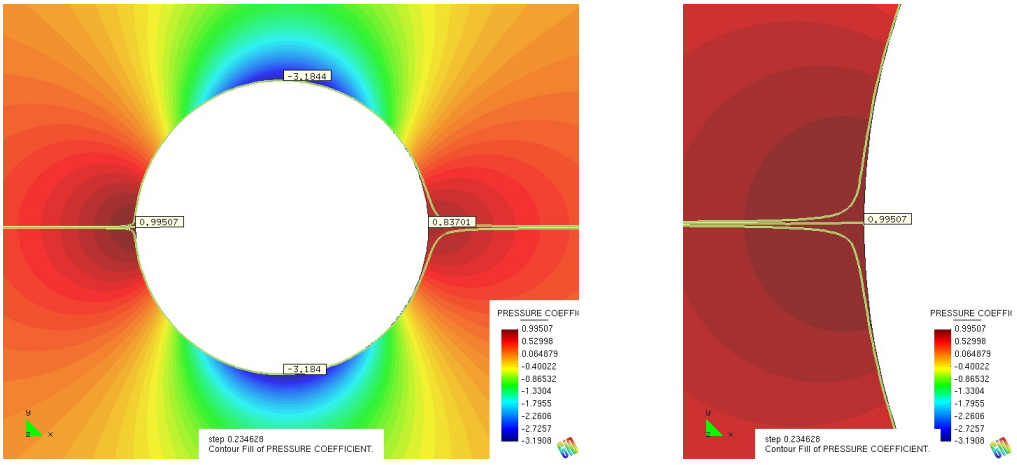
After a fairly short transitory period the stationary regime is reached in the incompressible case. At this point the upwinding term (first part of the stabilization term) in equation (3.63) becomes important to stabilize the steady state solution.

For the following analysis it is convenient to introduce the *pressure coefficient* C_p, a dimensionless number in fluid dynamics to describe the relative pressures throughout a flow field, that is defined as

$$C_p = \frac{p - p_\infty}{\frac{1}{2}\rho_\infty u_\infty^2} \tag{3.79}$$

where p is the pressure at the point of interest, p_∞ the freestream pressure, ρ_∞ the fluid density and u_∞ the freestream velocity of the fluid. In many situations in aerodynamics and hydrodynamics the pressure coefficient at a point near a body is independent of the body size. Consequently, respecting geometric and fluid flow similarities, an engineering model can be tested in a wind or water tunnel, pressure coefficients can be determined at critical locations and used with confidence to predict the fluid pressure around a full-size aircraft or boat.

Figure 3.6 shows a contour fill of the pressure coefficient around the cylinder. We will focus on the values of C_p on the boundary, where a slip condition was applied, as these are known from the analytical solution of the problem. The stagnation point with a theoretical value of $C_p = 1$ is met perfectly. On the top and at the bottom of the illustrated circle we obtain a slight deviation from $C_p = -3$, as well as after the cylinder where the solution is not exactly symmetric ($C_p < 1$). This indicates the numerical dissipation of the algorithm. Nevertheless, the results can be considered as very satisfactory.



(a) Pressure coefficient C_p and streamlines

(b) Stagnation point in detail

Figure 3.6: Incompressible airflow around a cylinder

It shall be remarked that the above results may be obtained either by choosing directly $\alpha = 0$ in the solver properties or by setting $Ma = 0.001$ in the Python script for example.

Compressible Flow

The compressible case has been tested with the freestream Mach number $Ma_\infty = 0.3$. In general this value is considered as limit for a flow to be treated still as incompressible. However, in the airflow around the cylinder higher Mach numbers appear locally.
This time the transitory period is much longer due to some oscillations caused by a propagating pressure wave,

- either starting from the inlet if the velcoity is zero in the whole domain at $t = 0$,
- or emerging around the cylinder if the velocity field is initialized with the freestream velocity u_∞ and its normal component is cut off by the slip condition.

Both possibilities have been tested; in the latter the relative velocity between wave propagation speed and freestream velocity before the cylinder has been checked.
The reflection of these waves at the boundaries shows that the conditions there are implemented correctly, so that, in the end, it takes some time until the numerical dissipation of the scheme copes with the mentioned oscillations of this test case. In a real simulation this is generally overcome by enlarging the distance between the object of study and the domain boundaries, characterized by a very fine mesh for the points of interest and a rather coarse grid at the distant boundary.

In compressible flow, and particularly in high-speed flow, the *dynamic* pressure $\frac{1}{2}\rho u^2$ is no longer an accurate measure of the difference between *stagnation* pressure and *static* pressure. Also the familiar relationship that *stagnation* pressure is equal to *total* pressure does not always hold true. As a result, pressure coefficients can be greater than one in compressible flow: $C_p > 1$ indicates that the freestream flow is supersonic ($Ma > 1$) implying the presence of shock waves (Wikipedia – The Free Encyclopedia). However, as mentioned in Section 3.2.1, we focus on isentropic flow of perfect gases in subsonic regime where the mentioned relations are always true.

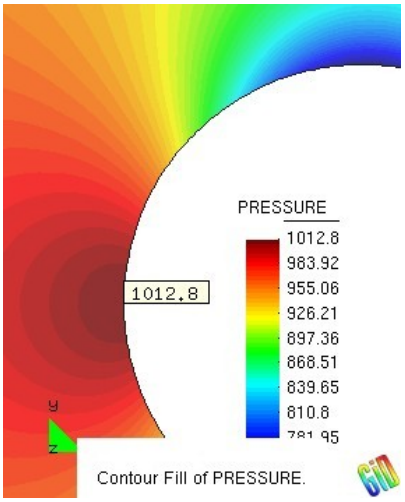


Figure 3.7: Compressible airflow around the cylinder - pressure at the stagnation point

Figure 3.7 shows a mean pressure value at the stagnation point near the steady-state solution. As there were still some oscillations the point evolution was averaged over a couple of time steps. On the other hand, the theoretical value was calculated by the relation

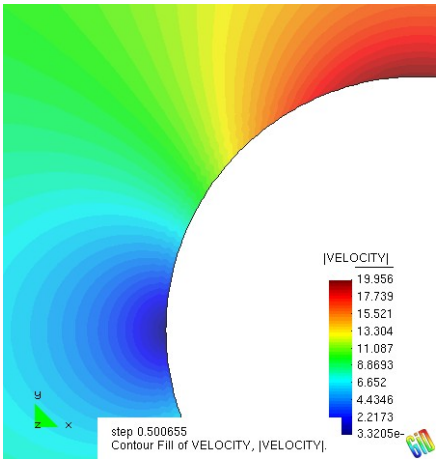
$$p_0 = p_\infty \left(1 + \frac{\gamma - 1}{2} Ma_\infty^2\right)^{\frac{\gamma}{\gamma - 1}}, \tag{3.80}$$

valid for the stagnation pressure p_0 in isentropic flow (Candel, 2005), and where the freestream pressure

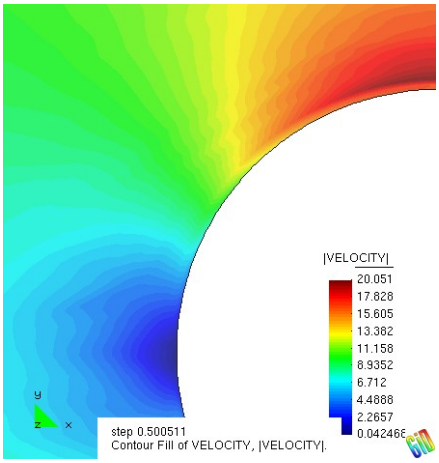
$$p_\infty = \frac{\rho_\infty}{\gamma} \left(\frac{u_\infty}{Ma_\infty}\right)^2. \tag{3.81}$$

is a function of the other freestream parameters – Mach number, velocity and density – defined at the beginning of the simulation run. The adiabatic exponent or ratio of specific heats $\gamma = 1.4$ for air. Thus, we finally obtain $p_0 \approx 1013,7Pa$ which sounds quite good compared with our numerical result $1012.8Pa$. The freestream pressure for comparison writes $p_\infty \approx 952.4Pa$.

In Section 3.4.4 we mentioned that several issues related to the uncoupling of velocity and pressure in fractional step schemes still deserve further analysis. One of these was the computed pressure near boundaries and the stability of the pressure itself. Even a slight instability in the pressure values may influence the velocity components severely.
Figure 3.8 shows a comparison that has been made with two different implementations of the convective term, both in conservative form, which should actually yield the same result. On the left-hand side (3.8(a)) the term has been implemented as a whole whereas on the right-hand (3.8(b)) side two linearized terms have been used. Until now no explanation has been found. Maybe this observation is related to the choice of the advective velocity $\boldsymbol{a}$ and the respective approximations in view of the edge-based data structure that might be critical when the linearization is done.



(a) Implementation of the conservative convection term as a whole



(b) Implementation of the conservative convective term in linearized form

Figure 3.8: Zoom on the boundary layer of the cylinder in compressible airflow

3.6.3. NACA 0012 Airfoil

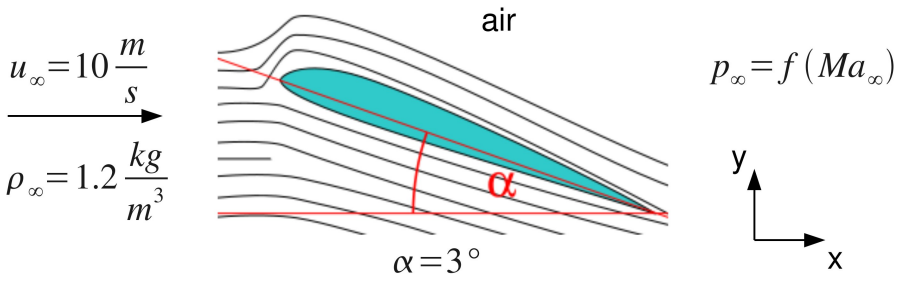


Figure 3.9: NACA 0012 airfoil – geometry and problem definition

The freeflow conditions known from the cylinder test case have been used on the NACA 0012 airfoil that has been inclined by the angle of attack $\alpha = 3°$. This time as well, a slip condition has been applied on the airfoil contour. As mentioned above, a very fine mesh has been created near the boundary layer whereas huge cells are employed far away from the airfoil. The grid of approximately 11.500 elements is shown in Figure 3.10.

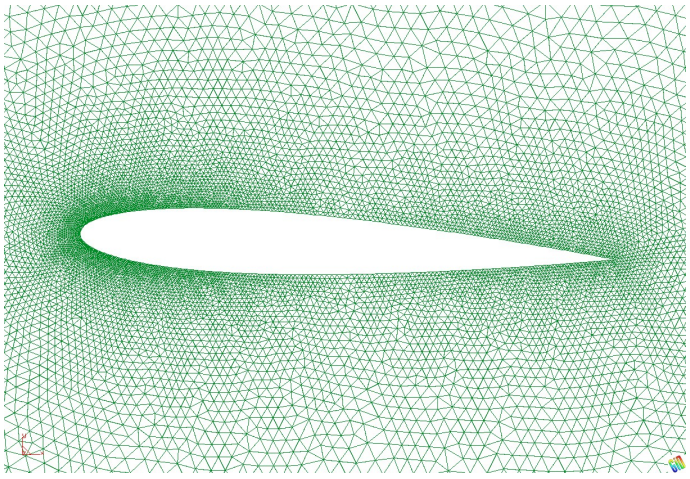


Figure 3.10: Fine mesh for the boundary layer of the airfoil

Incompressible Flow

As the pressure distribution on the contour is responsible for the lift force, the two corresponding dimensionless coefficients C_p and C_l are linked by an equation

$$C_l = \int_{LE}^{TE} \big(C_{pl}(x) - C_{pu}(x)\big)dx \tag{3.82}$$

where C_{pl} and C_{pu} are the pressure coefficients on lower and upper surface respectively, and the abbreviations LE and TE stand for leading and trailing edge of the airfoil.

For $Ma = 0$ the pressure coefficient C_p has been computed on the contour of the airfoil, using once more equation (3.79), and was plotted against the relative chord length. The numerical results are compared in Figure 3.11 with theoretical ones resulting from the potential theory. Apart from the peak in the pressure coefficient on the upper surface C_{pu},

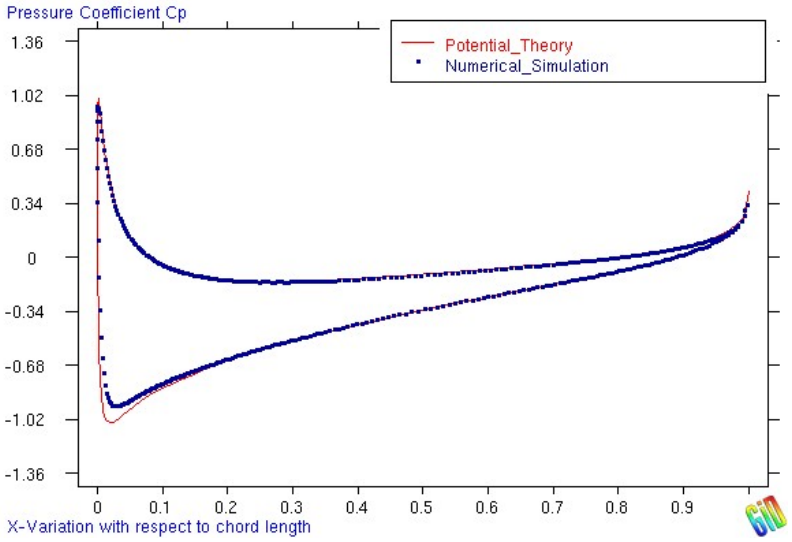


Figure 3.11: Pressure coefficient on the airfoil contour at $Ma = 0$

our results fit perfectly with the analytical solution. The slight discrepancy indicates that the mesh is not fine enough in this region. We can be quite sure of this as we will encounter a similar behaviour in the compressible case where the same mesh has been used.

A typical contour fill for the pressure coefficient is presented in Figure 3.12.

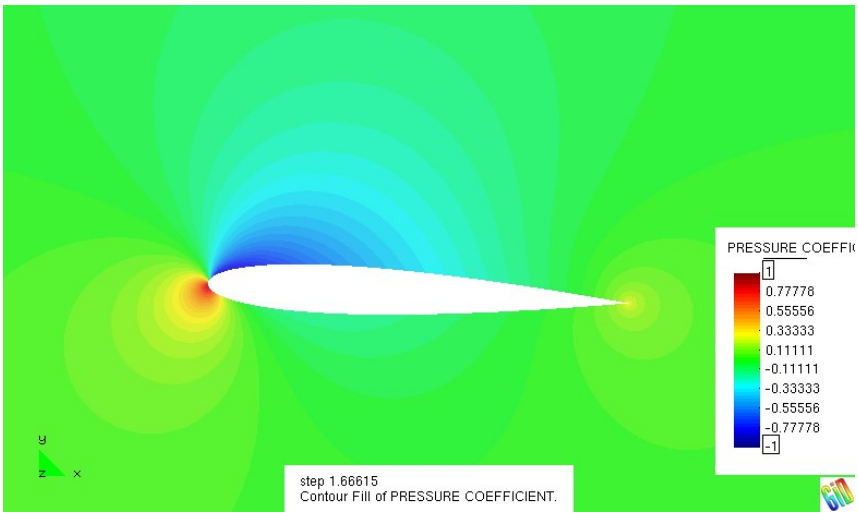
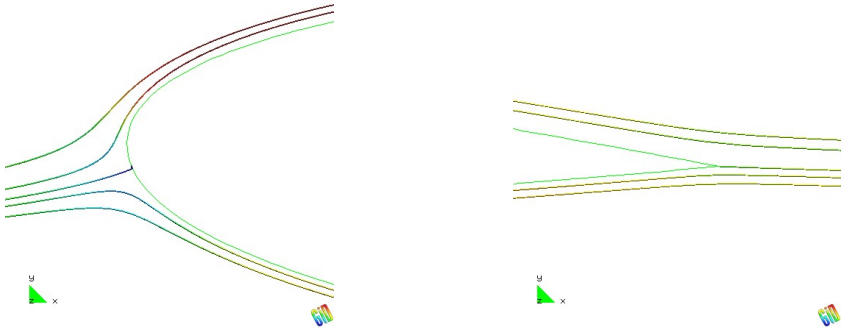


Figure 3.12: Typical contour fill of the pressure coefficient C_p at $Ma = 0$

Having a closer look on the leading edge of the airfoil (Figure 3.13(a)), we can state that the position of the stagnation point is correct as well as its value of the pressure coefficient ($C_p = 1$). Also the trailing edge looks great, especially because we are treating the node as an interior point (the normal defined there by the two adjacent faces is physically without any meaning so that an applied slip condition would probably distort the flow).



(a) Leading edge (b) Trailing edge

Figure 3.13: Streamlines on the NACA 0012 airfoil

Compressible Flow

For $Ma = 0.3$ the same analysis as in the incompressible case has been made. Analog to the result before, we suspect that the mesh is not fine enough near the pressure peak and therefore responsible for the slight discrepancy illustrated in Figure 3.14.

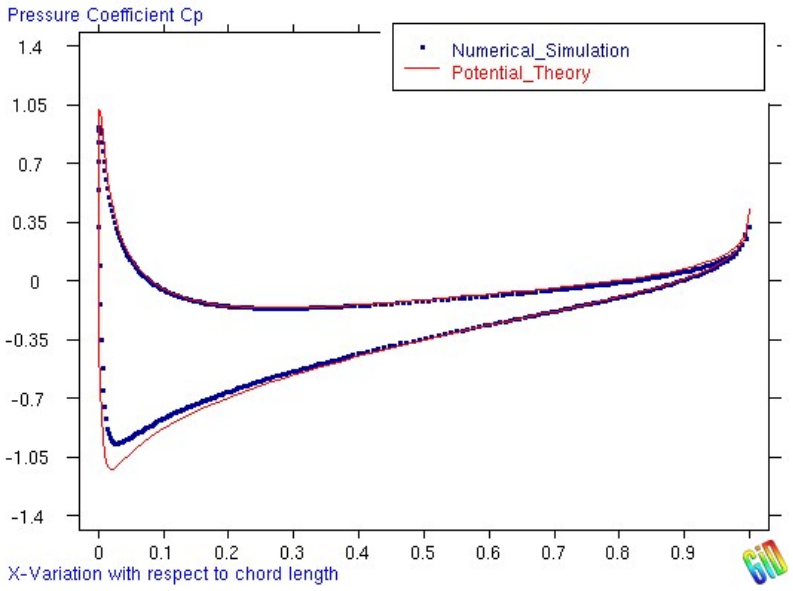


Figure 3.14: Pressure coefficient on the airfoil contour at $Ma = 0.3$

3.6.4. ONERA M6 Wing

As we only have treated two-dimensional examples so far, we will demonstrate with the ONERA M6 wing that our algorithm also works in three dimensions. Evidently, this section is characterized by a strong qualitative approach. We mainly wanted to know whether the implemented code is capable of tackling 3D cases. Figure 3.15 shows a contour fill of the pressure coefficient in the transitory period. We already can recognize the similarity to Figure 3.12 of the airfoil. The numerical artifacts in the picture can be traced back to a bug in GiD, which has already been corrected in the newest beta version of the software.

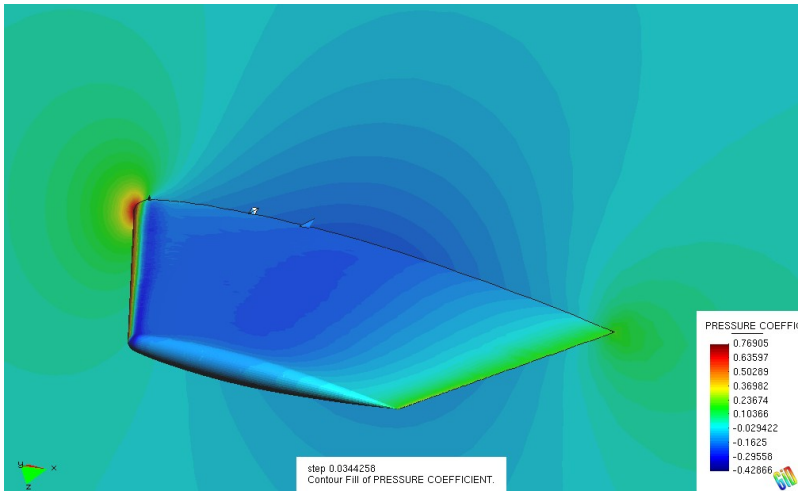


Figure 3.15: ONERA M6 wing – a tridimensional test case

We have seen before that a fine mesh is necessary in order to resolve the boundary layer correctly. This means that an adequate simulation of the ONERA wing becomes expensive in terms of computational effort, even for an edge-based implementation. That is why we scheduled the run and the comparison with experimental data of certain cut planes for the summer months, when the code will be completely parallelized and the CIMNE cluster will be working hopefully.

4 Preparation of Fluid-Structure Coupling

Having implemented and validated the general flow solver so far, this chapter will focus on the preparation of fluid-structure coupling and explain the necessary modifications in order to perform FSI simulations in the end.
First the principal solving procedures for coupled problems are elucidated.
Subsequently, the arbitrary Lagrangian-Eulerian description is derived from the classical kinematical viewpoints and its effect on the conservation equations of mass, momentum and energy is shown.
After some preliminary tests with moving meshes and the respective calculation of interface displacements and forces, expectations for the real fluid-structure coupling are presented.

4.1. Solving Procedures for Coupled Problems

There are several techniques for solving multi-disciplinary problems. Finding a suitable approach for each case highly depends on the category of the problem and the details of each field, especially for time-dependant problems. Concerning FSI this means that, depending on the nature of the fluid as well as on the properties of the structure, the characteristics of the coupling can be quite different. In this section an overview of Kratos' solving methodologies is given according to Dadvand (2007).

4.1.1. Sequential Solution

Considering the one-way coupled problem of Figure 2.2, the solving procedure is trivial. Since only the subsystem S_2 depends on u_1 (the solution of S_1), the problem can be solved easily by tackling the subsystem S_1 first to determine its solution u_1, which is used in turn to solve S_2. The in Figure 4.1 represented course of action is evaluated at each time step for transient problems.

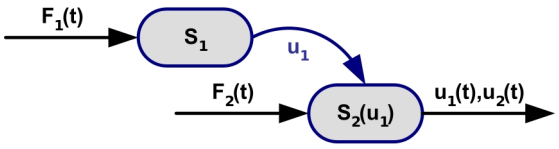


Figure 4.1: Sequential solution of a weakly coupled problem

Being $\mathcal{L}_i$ the operator applied over the domain of the respective subsystem S_i, the governing equations for the illustrated one-way coupled system can be written as

$$\begin{aligned} \mathcal{L}_1(u_1, t) &= f_1(t), \\ \mathcal{L}_2(u_1, u_2, t) &= f_2(t). \end{aligned} \tag{4.1}$$

Having applied temporal and spatial discretization within the scope of the finite element method, the following matrix system has to be solved at each time step:

$$\begin{bmatrix} \mathbf{K_{11}} & \mathbf{0} \\ \mathbf{K_{21}} & \mathbf{K_{22}} \end{bmatrix} \begin{Bmatrix} \mathsf{U}_1 \\ \mathsf{U}_2 \end{Bmatrix} = \begin{Bmatrix} \mathsf{f}_1(\mathsf{t}) \\ \mathsf{f}_2(\mathsf{t}) \end{Bmatrix}. \tag{4.2}$$

where $\mathbf{K_{11}}$ and $\mathbf{K_{22}}$ are the system matrices corresponding to the field variables of subsystems S_1 and S_2 respectively, and $\mathbf{K_{21}}$ represents the field system matrix corresponding to the interaction variables.

However, this rather easy approach called *sequential solution* is not possible for strongly coupled problems as shown in Figure 2.3. In this case either a monolithic approach or a staggered method has to be applied.

4.1.2. Monolithic Approach

By contrast, the *monolithic approach* treats the multi-disciplinary problem as a whole. The interacting fields are modeled together resulting in a coupled continuous model, which is solved directly in one step. Figure 4.2 illustrates this procedure assuming that there are only two subsystems.

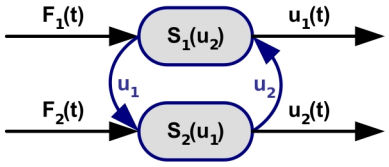


Figure 4.2: Monolithic scheme for a strongly coupled problem

Using the same notation as above, the governing equations for the now considered two-way coupled system can be written as

$$\begin{aligned} \mathcal{L}_1(u_1, u_2, t) &= f_1(t), \\ \mathcal{L}_2(u_1, u_2, t) &= f_2(t). \end{aligned} \tag{4.3}$$

After temporal and spatial discretization the following matrix system is obtained not allowing a sequential solution any more:

$$\begin{bmatrix} \mathbf{K_{11}} & \mathbf{K_{12}} \\ \mathbf{K_{21}} & \mathbf{K_{22}} \end{bmatrix} \begin{Bmatrix} U_1 \\ U_2 \end{Bmatrix} = \begin{Bmatrix} f_1(t) \\ f_2(t) \end{Bmatrix}. \tag{4.4}$$

This time the interaction matrices $\mathbf{K_{12}}$ and $\mathbf{K_{21}}$ couple the field variables of the two subsystems in a manner that demands the problem to be solved at once.

Though this approach seems to be very easy and natural, several difficulties are encountered in practice:

- *difficulty of the formulation*: The multi-disciplinary continuous models are usually complex by nature making the discretization process a tedious task.
- *size and bandwidth of the problem*: The obligation of solving all fields simultaneously renders the monolithic approach expensive in terms of memory and cpu performance.
- *implementation cost*: The interaction between different fields requires the interface matrices to be customized to reflect the new variables. As generally severe modifications are part of this adaptation, a re-use of these matrices is nearly impossible.

Despite all mentioned disadvantages, one should not forget that a monolithic approach perfectly models the interaction between the different fields and results in a more robust and more stable formulation for solving coupled problems.

4.1.3. Staggered Methods

The intention of *staggered methods* is to solve each field separately and thus use less resources than the monolithic approach. In each step only one part of the problem is solved, which is a great advantage in the solution of large problems. The interaction is assured by applying certain techniques for transforming variables from one field to another. Some of these techniques are outlined here:

Predicition consists of predicting the value of the dependent variables in the next time step. As shown in Figure 4.3(a) the predicted variable u_{2P}^{n+1} is used to solve the subsystem S_1 separately, which decouples different fields in problems with strong coupling. Common choices are:

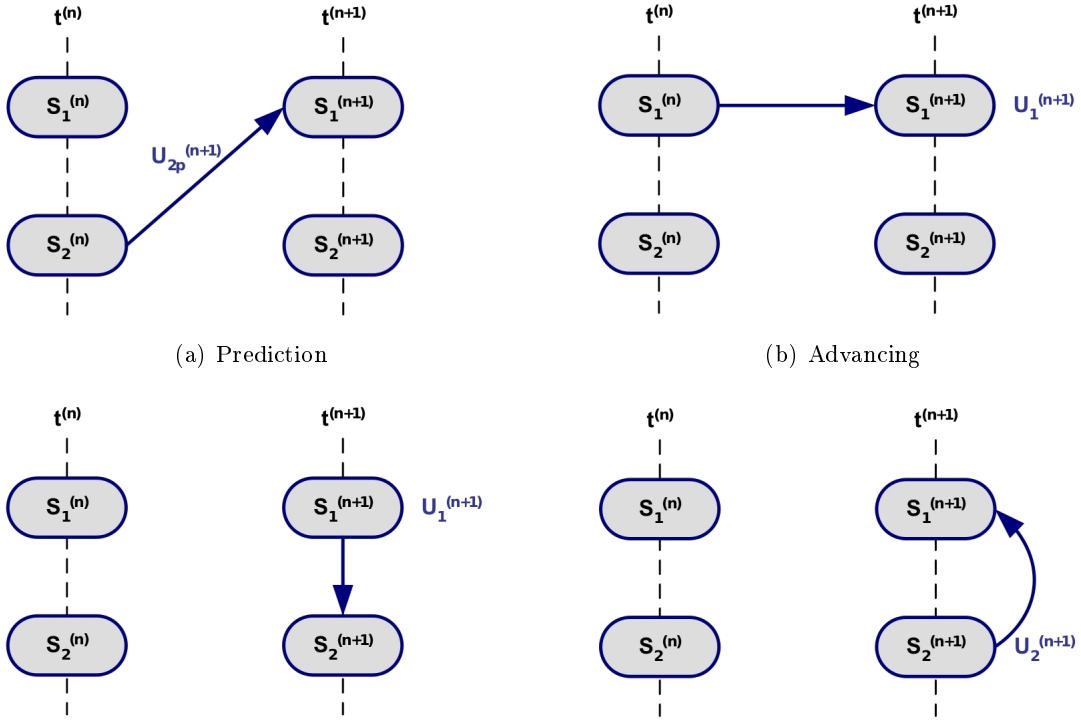
- the last-solution predictor: $u_p^{n+1} = u^n$
- or prediction by solution gradient: $u_p^{n+1} = u^n + \Delta t \cdot \dot{u}^n$

where $\Delta t = t^{n+1} - t^n$ and $\dot{u}^n = \left(\frac{\partial u}{\partial t}\right)^n$.

Advancing means calculating the next time step of a subsystem using the calculated or predicted solution of other subsystems. This technique is illustrated in Figure 4.3(b).

Substitution is a trivial technique shown in Figure 4.3(c) that uses the calculated value of one field in another field to solve it separately.

Correction substitutes u_2^{n+1} in place of the predicted value u_{2p}^{n+1} and solves again the subsystem S_1 to obtain a better result. This implies that the subsystem S_1 has been solved introducing the predicted value u_{2p}^{n+1} and that the here obtained result u_2^{n+1} has been used to advance in turn subsystem S_2 in order to calculate u_2^{n+1}. Obviously this procedure shown in Figure 4.3(d) can be repeated more than once.



(a) Prediction (b) Advancing

(c) Substitution (d) Correction

Figure 4.3: Techniques for staggered methods

A staggered method can be planned using the techniques presented above. Returning to the problem of Figure 2.3, the following course of action, illustrated in Figure 4.4, would be possible:

1. Predicition: $u_p^{n+1} = u_2^n + \Delta t \cdot \dot{u}_2^n$
2. Advancing: $S_1^{n+1}\left(u_p^{n+1}\right) \rightarrow u_1^{n+1}$
3. Substitution: $u_1^{n+1} = u_1^{n+1}$ for S_2
4. Advancing: $S_2^{n+1}\left(u_1^{n+1}\right) \rightarrow u_2^{n+1}$

As one may guess, staggered methods require a careful formulation to avoid instabilities and to obtain an accurate solution, thus increasing the attention concerning modeling. Nevertheless there are some important advantages:

- The definition of *different discretizations for each field* is possible, with varying mesh characteristics if necessary.

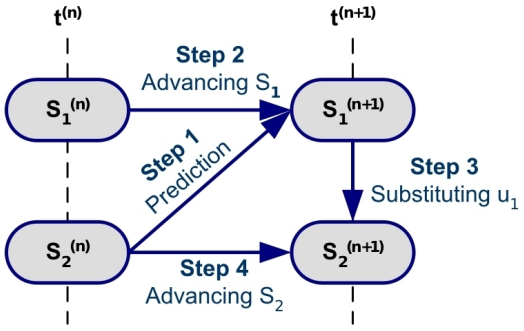


Figure 4.4: A typical staggered method for solving a two-way coupled system

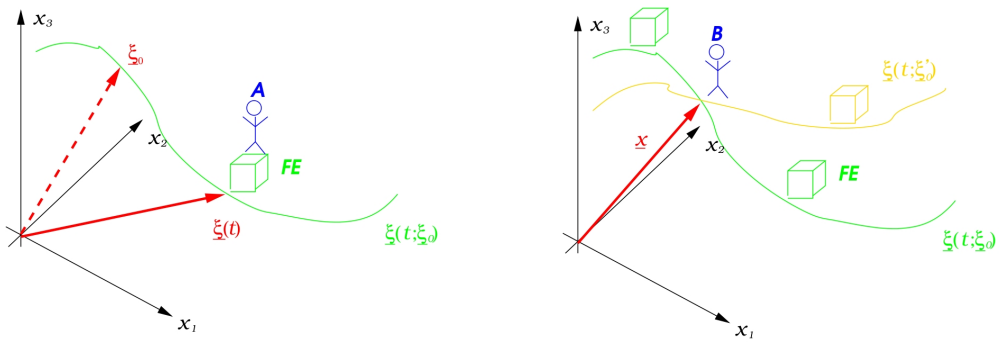
- *Existing single field codes may be re-used* for solving multi-disciplinary problems almost without modification.
- Solving only one part of the problem does not only *use less resources* compared to monolithic schemes; it also comes up with the idea of segmenting the solving procedure for large single field problems and *scheduling the algorithm in parallel.*

4.2. Arbitrary Lagrangian-Eulerian Description

The numerical simulation of multi-dimensional problems in fluid dynamics and nonlinear solid mechanics often requires coping with strong distortions of the continuum under consideration while simultaneously assuring a clear delineation of free surfaces and fluid-fluid, solid-solid or fluid-structure interfaces. To deal with large distortions and to provide an accurate resolution of material interfaces and mobile boundaries, an appropriate kinematical description is fundamental.
The *arbitrary Lagrangian-Eulerian* (ALE) description was developed in an attempt to combine the advantages of the classical kinematical descriptions while minimizing their respective drawbacks as far as possible. This is the reason why this chapter starts with a brief reminder of the classical approaches: the *Lagrangian* and the *Eulerian* point of view (Donea et al., 2004; Adams, 2007).

4.2.1. Lagrangian vs. Eulerian Description

Describing a continuum in motion, there are two primary ways of doing so. The first one is to pick a specific particle, denoted as fluid element ***FE*** in Figure 4.5, and to follow its movement $\boldsymbol{\xi}(t, \boldsymbol{\xi_0})$ in the course of time. Observer ***A*** in Figure 4.5(a) represents this Lagrangian viewpoint. Observer ***B*** however is situated in a fixed position $\boldsymbol{x}$ of the spatial domain. From his Eulerian point of view he monitors the different fluid elements passing by, $\boldsymbol{\xi_0}$ and $\boldsymbol{\xi_0'}$ in Figure 4.5(b).



(a) Lagrangian viewpoint (b) Eulerian viewpoint

Figure 4.5: Comparison of classical kinematical approaches

In *Lagrangian algorithms* – mainly used in structural mechanics – each individual node of the computational mesh follows the associated material particle during motion as shown in the upper part of Figure 4.6. Obviously, this facilitates the tracking of free surfaces and of interfaces between different materials. Moreover, materials with history-dependent constitutive relations can be treated easily since each finite element of a Lagrangian mesh always contains the same material particles. Its weakness is its inability to follow large distortions of the computational domain without recourse to frequent remeshing operations.

In *Eulerian algorithms* – widely used in fluid dynamics – the computational mesh is fixed and the continuum moves with respect to the grid. As time evolves, the physical quantities associated with the fluid particles passing through the fixed region of space, are examined. This implies a relatively easy handling of large distortions in the continuum motion. However, interface definitions and the resolution of flow details are generally less precise.

4.2.2. ALE – Generalization of both Approaches

Since the ALE description of motion is a generalization of the Lagrangian and Eulerian descriptions, the nodes of the computational mesh may be

- moved with the continuum in normal Lagrangian fashion,
- held fixed in Eulerian manner or
- moved in some arbitrarily specified way as shown in Figure 4.6

to give a continuous rezoning capability.

In the ALE description of motion that will be derived now according to Donea et al. (2004), neither the material domain $R_{\boldsymbol{X}}$ made up of material particles $\boldsymbol{X}$ nor the spatial domain $R_{\boldsymbol{x}}$ consisting of spatial points $\boldsymbol{x}$ is taken as a reference. Instead a third domain is introduced: the referential configuration $R_{\boldsymbol{\chi}}$ with reference coordinates $\boldsymbol{\chi}$ identifying the grid points. Figure 4.7 shows these domains and the one-to-one transformations relating the configurations. The referential domain $R_{\boldsymbol{\chi}}$ is mapped into the material and spatial

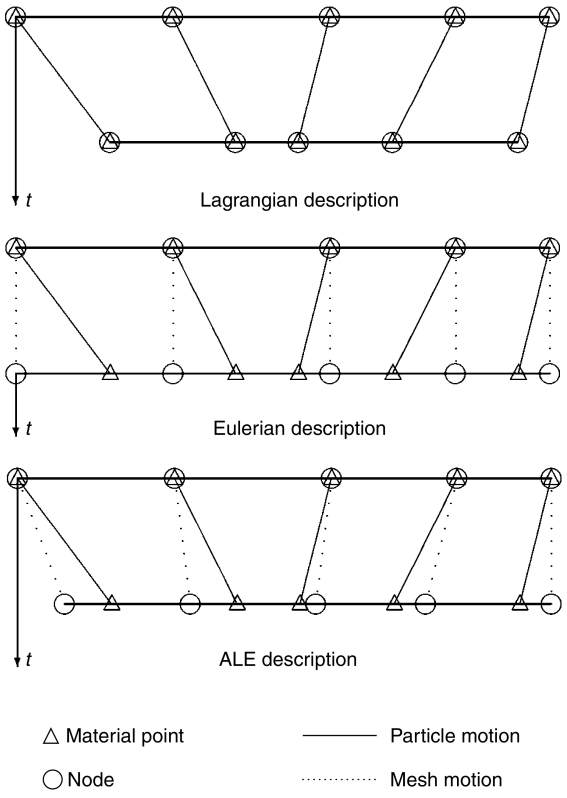


Figure 4.6: Lagrangian, Eulerian and ALE mesh and particle motion in one dimension

domains by $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$ respectively. The particle motion $\boldsymbol{\varphi}$ may then be expressed as $\boldsymbol{\varphi} = \boldsymbol{\Phi} \circ \boldsymbol{\Psi}^{-1}$, clearly showing the dependency of the three mappings:

- The application $\boldsymbol{\varphi}$ represents the mapping from the material domain $R_{\boldsymbol{X}}$ to the spatial domain $R_{\boldsymbol{x}}$, relating the motion of the material points $\boldsymbol{X}$ to the spatial coordinates $\boldsymbol{x}$. It is defined such that

$$\begin{aligned} \boldsymbol{\varphi} : R_{\boldsymbol{X}} \times [t_0, t_{end}[&\longrightarrow R_{\boldsymbol{x}} \times [t_0, t_{end}[\\ (\boldsymbol{X}, t) &\longmapsto \boldsymbol{\varphi}(\boldsymbol{X}, t) = (\boldsymbol{x}, t) \end{aligned} \tag{4.5}$$

  which allows us to link $\boldsymbol{X}$ and $\boldsymbol{x}$ in time by the law of motion, namely

$$\boldsymbol{x} = \boldsymbol{x}(\boldsymbol{X}, t). \tag{4.6}$$

  The physical time is measured by the same variable t in both domains so that for every fixed instant t, the mapping $\boldsymbol{\varphi}$ defines a configuration in the spatial domain. It is convenient to employ a matrix representation for its gradient

$$\frac{\partial \boldsymbol{\varphi}}{\partial (\boldsymbol{X}, t)} = \begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} & \boldsymbol{u} \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{4.7}$$

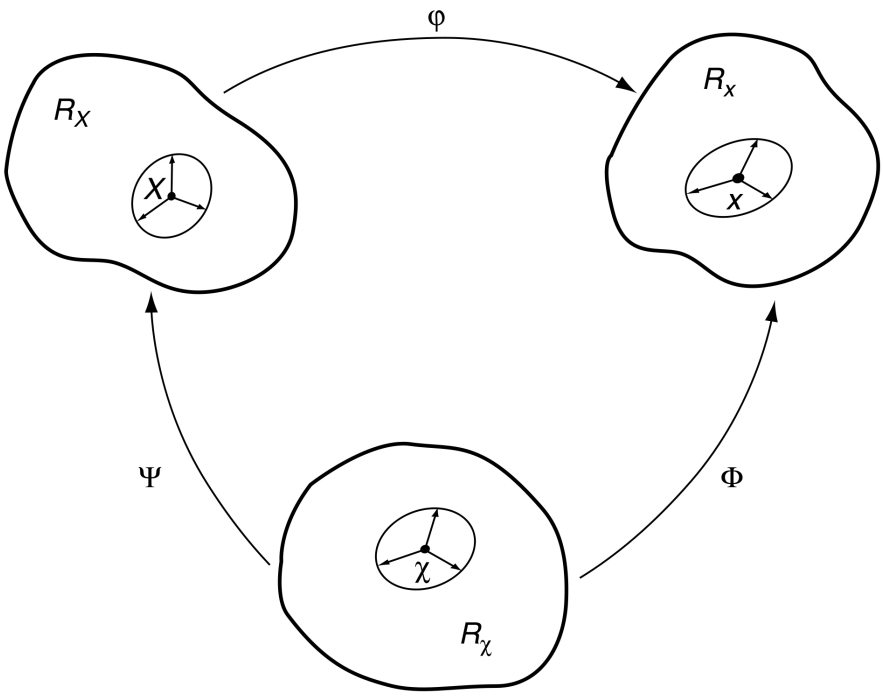


Figure 4.7: Transformations between material, spatial and referential configuration

where $\mathbf{0}^T$ is a null row-vector and the *material velocity* $\boldsymbol{u}$ is

$$\boldsymbol{u}(\boldsymbol{X},t) = \left.\frac{\partial \boldsymbol{x}}{\partial t}\right|_{\boldsymbol{X}} \tag{4.8}$$

with $\left|_{\boldsymbol{X}}\right.$ indicating that the respective coordinate, material in this case, is hold fixed.

- The mapping of $\boldsymbol{\Phi}$ from the referential domain $R_{\boldsymbol{\chi}}$ to the spatial domain $R_{\boldsymbol{x}}$ can be understood as the motion of the grid points in the spatial domain. It is represented by

$$\begin{aligned} \boldsymbol{\Phi} : R_{\boldsymbol{\Psi}} \times [t_0, t_{end}[&\longrightarrow R_{\boldsymbol{x}} \times [t_0, t_{end}[\\ (\boldsymbol{\chi}, t) &\longmapsto \boldsymbol{\Phi}(\boldsymbol{\chi}, t) = (\boldsymbol{x}, t) \end{aligned} \tag{4.9}$$

and its gradient is

$$\frac{\partial \boldsymbol{\Phi}}{\partial(\boldsymbol{\chi}, t)} = \begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} & \boldsymbol{u}_{mesh} \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{4.10}$$

where now, the *mesh velocity*

$$\boldsymbol{u}_{mesh}(\boldsymbol{\chi},t) = \left.\frac{\partial \boldsymbol{x}}{\partial t}\right|_{\boldsymbol{\chi}} \tag{4.11}$$

is involved. Note that both the material and the mesh move with respect to the laboratory. Thus, the corresponding material and mesh velocities have been defined

by deriving the equations of material motion and mesh motion in each case with respect to time.

- Finally, regarding the mapping of $\boldsymbol{\Psi}$ from the referential domain $R_{\boldsymbol{\chi}}$ to the material domain $R_{\boldsymbol{X}}$, it is more convenient to represent directly its inverse $\boldsymbol{\Psi}^{-1}$

$$\begin{aligned} \boldsymbol{\Psi}^{-1} : R_{\boldsymbol{X}} \times [t_0, t_{end}[&\longrightarrow R_{\boldsymbol{\chi}} \times [t_0, t_{end}[\\ (\boldsymbol{X}, t) &\longmapsto \boldsymbol{\Psi}^{-1}(\boldsymbol{X}, t) = (\boldsymbol{\chi}, t) \end{aligned} \tag{4.12}$$

and its gradient is

$$\frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X}, t)} = \begin{pmatrix} \dfrac{\partial \boldsymbol{\chi}}{\partial \boldsymbol{X}} & \boldsymbol{u}_{ref} \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{4.13}$$

where the velocity $\boldsymbol{u}_{ref}$ is defined as

$$\boldsymbol{u}_{ref} = \left.\frac{\partial \boldsymbol{\chi}}{\partial t}\right|_{\boldsymbol{X}}. \tag{4.14}$$

It can be interpreted as the *particle velocity in the referential domain* since it measures the time variation of the referential coordinate $\boldsymbol{\chi}$ holding the material particle $\boldsymbol{X}$ fixed.

The relation between the velocities $\boldsymbol{u}$, $\boldsymbol{u}_{mesh}$ and $\boldsymbol{u}_{ref}$ can be obtained by differentiating the relation $\boldsymbol{\varphi} = \boldsymbol{\Phi} \circ \boldsymbol{\Psi}^{-1}$:

$$\begin{aligned} \frac{\partial \boldsymbol{\varphi}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) &= \frac{\partial \boldsymbol{\Phi}}{\partial(R_{\boldsymbol{\chi}}, t)}\left(\boldsymbol{\Psi}^{-1}(\boldsymbol{X}, t)\right) \quad \frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) \\ &= \frac{\partial \boldsymbol{\Phi}}{\partial(R_{\boldsymbol{\chi}}, t)}(\boldsymbol{\chi}, t) \quad \frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X}, t)}(\boldsymbol{X}, t) \end{aligned} \tag{4.15a}$$

or, in matrix format,

$$\begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{X}} & \boldsymbol{u} \\ \mathbf{0}^T & 1 \end{pmatrix} = \begin{pmatrix} \dfrac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} & \boldsymbol{u}_{mesh} \\ \mathbf{0}^T & 1 \end{pmatrix} \begin{pmatrix} \dfrac{\partial \boldsymbol{\chi}}{\partial \boldsymbol{X}} & \boldsymbol{u}_{ref} \\ \mathbf{0}^T & 1 \end{pmatrix} \tag{4.15b}$$

which yields, after block multiplication, $\boldsymbol{u} = \boldsymbol{u}_{mesh} + \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} \cdot \boldsymbol{u}_{ref}$. Introducing the *convective velocity* $\boldsymbol{u}_{conv}$ as the relative velocity between material and mesh, this equation can be rewritten as

$$\boldsymbol{u}_{conv} := \boldsymbol{u} - \boldsymbol{u}_{mesh} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\chi}} \cdot \boldsymbol{u}_{ref}. \tag{4.16}$$

The convective velocity can be interpreted as the particle velocity relative to the mesh as seen from the spatial domain $R_{\boldsymbol{x}}$.

Note that both Lagrangian and Eulerian formulations may be obtained as particular cases of this generalized approach:

- Choosing $\boldsymbol{\Psi} = \boldsymbol{I}$ (where $\boldsymbol{I}$ is the identity application), referential domain $R_{\boldsymbol{\chi}}$ and material domain $R_{\boldsymbol{X}}$ coincide ($\boldsymbol{\chi} \equiv \boldsymbol{X}$) resulting in a *Lagrangian description*. Material and mesh velocity, equations (4.8) and (4.11), are equal leading to a convective velocity of zero (see equation (4.16)) and thus preventing the presence of convective terms in the conservation laws.
- With the choice of $\boldsymbol{\Phi} = \boldsymbol{I}$ on the other hand, referential domain $R_{\boldsymbol{\chi}}$ and spatial domain $R_{\boldsymbol{x}}$ coincide ($\boldsymbol{\chi} \equiv \boldsymbol{x}$) conducting to an *Eulerian description*. The mesh velocity $\boldsymbol{u}_{mesh}$ obtained from equation (4.11) is zero whereas the convective velocity $\boldsymbol{u}_{conv}$ is identical to the material velocity $\boldsymbol{u}$.

4.2.3. ALE Form of Conservation Equations

In order to express the conservation laws for mass, momentum and energy in an ALE framework, a relation between the material (or total) time derivative, which is inherent in conservation laws, and the referential time derivative is needed.

Fundamental ALE Relation

Therefore a scalar physical quantity, denoted by $f(\boldsymbol{x},t)$, $f^*(\boldsymbol{\chi},t)$ and $f^{**}(\boldsymbol{X},t)$ in the spatial, referential and material domain respetively, is considered in the following. Using the mapping properties of $\boldsymbol{\Psi}$, the transformation from the referential description $f^*(\boldsymbol{\chi},t)$ of the scalar physical quantity to its material description $f^{**}(\boldsymbol{X},t)$ is cast as

$$f^{**} = f^* \circ \boldsymbol{\Psi}^{-1} \tag{4.17a}$$

or

$$f^{**}(\boldsymbol{X},t) = f(\boldsymbol{\Psi}^{-1}(\boldsymbol{X},t),t). \tag{4.17b}$$

The gradient of this expression can be computed as

$$\frac{\partial f^{**}}{\partial(\boldsymbol{X},t)}(\boldsymbol{X},t) = \frac{\partial f^*}{\partial(\boldsymbol{\chi},t)}(\boldsymbol{\chi},t) \quad \frac{\partial \boldsymbol{\Psi}^{-1}}{\partial(\boldsymbol{X},t)}(\boldsymbol{X},t) \tag{4.18a}$$

which is amenable to the matrix form

$$\begin{pmatrix} \frac{\partial f^{**}}{\partial \boldsymbol{X}} & \frac{\partial f^{**}}{\partial t} \end{pmatrix} = \begin{pmatrix} \frac{\partial f^*}{\partial \boldsymbol{\chi}} & \frac{\partial f^*}{\partial t} \end{pmatrix} \begin{pmatrix} \frac{\partial \boldsymbol{\chi}}{\partial \boldsymbol{X}} & \boldsymbol{u}_{ref} \\ \boldsymbol{0}^T & 1 \end{pmatrix} \tag{4.18b}$$

Apart from the obvious statement $\frac{\partial f^{**}}{\partial \boldsymbol{X}} = \frac{\partial f^*}{\partial \boldsymbol{\chi}} \frac{\partial \boldsymbol{\chi}}{\partial \boldsymbol{X}}$, block multiplication also arouses the desired relation between material and spatial time derivatives:

$$\frac{\partial f^{**}}{\partial t} = \frac{\partial f^*}{\partial t} + \frac{\partial f^*}{\partial \boldsymbol{\chi}} \cdot \boldsymbol{u}_{ref} \tag{4.19}$$

Taking into account that, in fluids, constitutive relations are naturally expressed in the spatial configuration and the Cauchy stress tensor is the natural measure for stresses, it is

more convenient to rearrange the previous equation. Using equation (4.16) the gradient of the considered quantity is evaluated in the spatial domain instead of the referential one:

$$\frac{\partial f^{**}}{\partial t} = \frac{\partial f^{*}}{\partial t} + \frac{\partial f}{\partial \boldsymbol{x}} \cdot \boldsymbol{u}_{conv} \tag{4.20}$$

Dropping the stars, reveals the *fundamental ALE relation* in the end:

$$\begin{aligned} \left.\frac{\partial f}{\partial t}\right|_{\boldsymbol{X}} &= \left.\frac{\partial f}{\partial t}\right|_{\boldsymbol{\chi}} + \frac{\partial f}{\partial \boldsymbol{x}} \cdot \boldsymbol{u}_{conv} \\ &= \left.\frac{\partial f}{\partial t}\right|_{\boldsymbol{\chi}} + \boldsymbol{u}_{conv} \cdot \boldsymbol{\nabla} f \end{aligned} \tag{4.21}$$

It shows that the time derivative of the physical quantity f for a given particle $\boldsymbol{X}$, its material derivative, can be expressed as the sum of its local derivative (with the reference coordinate $\boldsymbol{\chi}$ held fixed) and a convective term considering the relative velocity $\boldsymbol{u}_{conv}$ between the material and the reference system.

Basic Conservation Equations

In order to obtain the ALE form of the conservation equations presented in Chapter 3, the material velocity $\boldsymbol{u}$ has to be replaced by exactly this convective velocity $\boldsymbol{u}_{conv}$ – defined by equation (4.16) – in the various convective terms. For convenience and to establish the link to the notation chosen in Section 3.2, the material time derivative is further denoted as

$$\frac{d}{dt} := \left.\frac{\partial}{\partial t}\right|_{\boldsymbol{X}} \tag{4.22a}$$

and the spatial time derivative as

$$\frac{\partial}{\partial t} := \left.\frac{\partial}{\partial t}\right|_{\boldsymbol{x}}. \tag{4.22b}$$

Recalling the conservation equations for mass (3.4b) and momentum (3.10b) in terms of the total time derivative

$$\frac{d\rho}{dt} = \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{x}} + \boldsymbol{u} \cdot \boldsymbol{\nabla}\rho = -\rho \boldsymbol{\nabla} \cdot \boldsymbol{u} \qquad \text{(mass)} \tag{4.23a}$$

$$\rho \frac{d\boldsymbol{u}}{dt} = \rho \left(\left.\frac{\partial \boldsymbol{u}}{\partial t}\right|_{\boldsymbol{x}} + (\boldsymbol{u} \cdot \boldsymbol{\nabla})\boldsymbol{u} \right) = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho \boldsymbol{g} \qquad \text{(momentum)} \tag{4.23b}$$

and introducing the convective velocity $\boldsymbol{u}_{conv}$ leads to the ALE differential forms:

$$\frac{d\rho}{dt} = \left.\frac{\partial \rho}{\partial t}\right|_{\boldsymbol{\chi}} + \boldsymbol{u}_{\boldsymbol{conv}} \cdot \boldsymbol{\nabla}\rho = -\rho \boldsymbol{\nabla} \cdot \boldsymbol{u} \qquad \text{(mass)} \tag{4.24a}$$

$$\rho \frac{d\boldsymbol{u}}{dt} = \rho \left(\left.\frac{\partial \boldsymbol{u}}{\partial t}\right|_{\boldsymbol{\chi}} + (\boldsymbol{u}_{\boldsymbol{conv}} \cdot \boldsymbol{\nabla})\boldsymbol{u} \right) = \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} + \rho \boldsymbol{g} \qquad \text{(momentum)} \tag{4.24b}$$

It is important to note that the right-hand side of equations (4.24) is written in classical spatial form, so to speak Eulerian, whereas the arbitrary motion of the computational mesh is only reflected on the left-hand side. For our implementation in particular, this means that the advective velocity has to be set to $\boldsymbol{a}_{ALE} := \boldsymbol{a} - \boldsymbol{u}_{mesh}$.
For the ALE form of the conservation equations for energy (total respectively internal), please refer to Donea et al. (2004).

Furthermore, one has to bear in mind that the mesh motion may increase or decrease the convection effects. Due to the already mentioned lack of stability of the standard Galerkin formulation in convection-dominated situations, these might influence the employed stabilization technique described in Section 3.4.4.

Boundary Conditions

In fact, boundary conditions are related to the problem, not to the description employed. Thus, the same boundary conditions as in Eulerian and Lagrangian descriptions are used. As the ALE formulation allows an accurate treatment of material surfaces, the following two conditions are required on a material surface:

- no particles can cross it and
- stresses must be continuous across the surface (if a net force is applied to a surface of zero mass, the acceleration is infinite).

In the case of fluid-structure interaction the particle velocity along solid-wall boundaries is coupled to the rigid or flexible structure. The enforcement of the kinematic requirement that no particle can cross the interface writes

$$\boldsymbol{u}_{ref} \cdot \boldsymbol{n} = 0 \quad \text{or} \quad \boldsymbol{u} \cdot \boldsymbol{n} = \boldsymbol{u}_{mesh} \cdot \boldsymbol{n} \tag{4.25}$$

where $\boldsymbol{n}$ indicates once more the outward unit normal.

However, due to the coupling between fluid and structure, extra conditions are needed to ensure that the fluid and structural domains will not detach or overlap during the motion. These coupling conditions depend on the fluid. An *inviscid fluid*, as no shear effects are considered, is free to slip along the structural interface. That is why only normal components are taken into account for the coupling:

$$\boldsymbol{d} \cdot \boldsymbol{n} = \boldsymbol{d}_{struct} \cdot \boldsymbol{n} \qquad \text{(continuity of normal displacements)} \tag{4.26a}$$

$$\boldsymbol{u} \cdot \boldsymbol{n} = \boldsymbol{u}_{struct} \cdot \boldsymbol{n} \qquad \text{(continuity of normal velocities)} \tag{4.26b}$$

Apart from these kinematical conditions, the dynamic condition claiming stresses in the fluid and stresses in the structure to be equal has to be verified:

$$-p\boldsymbol{n} = \boldsymbol{\sigma}_{struct} \cdot \boldsymbol{n} \qquad \text{(equality of stresses)} \tag{4.26c}$$

For the respective interface conditions of *viscous* fluids, e.g. in the aim of facilitating a future expansion of the implemented application, please refer to Donea et al. (2004) where they are outlined as well.

4.3. Preliminary Tests

In this section we will perform some "quasi" FSI simulations. This means that we perform tests on moving meshes whose motion is defined by algebraic functions and *not* by the coupling with a structural application.
As we prescribe the motion of the grid points by an algebraic function, it seems logical to calculate the mesh velocity $\boldsymbol{u}_{mesh}$ by

$$\boldsymbol{u}_{mesh}^{n+\frac{1}{2}} = \frac{\boldsymbol{x}^{n+1} - \boldsymbol{x}^{n}}{t^{n+1} - t^{n}} \tag{4.27a}$$

where $\boldsymbol{x}^n$ and $\boldsymbol{x}^{n+1}$ are the nodal position vectors at time level n and $n+1$ respectively. Nevertheless, we calculate $\boldsymbol{u}_{mesh}$ by the bias of the nodal displacements

$$\boldsymbol{u}_{mesh}^{n+\frac{1}{2}} = \frac{\boldsymbol{d}^{n+1} - \boldsymbol{d}^{n}}{t^{n+1} - t^{n}} \tag{4.27b}$$

where $\boldsymbol{d}^n = \boldsymbol{x}^n - \boldsymbol{x}^0$ refers to the initial geometry, because this is the way we will have to do it once the flow solver is coupled with a structural application.

Moreover, a moving mesh implies changes of the geometry data of its elements. This means that the edge data has to be recomputed by the function `MatrixContainer.BuildCSRData` whenever the grid points change their position. Certainly, this causes the edge-based implementation to lose its advantage of pre-computing integral data in weakly coupled problems where one iteration between flow and structural solver is sufficient. In problems with strong coupling however, it absolutely makes sense as many iterations between the two solvers will be necessary until the convergence of the solution within a specific time step is reached.

4.3.1. Geometric Conservation Law

A very common test in this context is the *geometric conservation law* for unsteady flow computations on moving and deforming finite element or finite volume grids.
The basic requirement is that any ALE computational method should be able to predict exactly the trivial solution of a uniform flow. The ALE equation of mass balance (4.24a) is usually taken as the starting point for the derivation of the geoemtric conservation law. The Reynolds transport theorem is used once again, this time applied to an arbitrary volume $V_m(t)$ whose boundary $A_m(t) = \partial V_m(t)$ moves with the mesh velocity $\boldsymbol{u}_{mesh}$:

$$\left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{V_m(t)} f(\boldsymbol{x},t) dV = \int_{V_m(t)} \left.\frac{\partial f(\boldsymbol{x},t)}{\partial t}\right|_{\boldsymbol{x}} dV + \int_{A_m(t)} f(\boldsymbol{x},t)\boldsymbol{u}_{mesh} \cdot \boldsymbol{n} dA \tag{4.28}$$

where, in this case, we have explicitly indicated that the time derivative in the first term of the right-hand side is a spatial time derivative, as in expression (3.2). Replacing the

scalar $f(\boldsymbol{x}, t)$ by the fluid density ρ and substituting the spatial time derivative $\partial f/\partial t$ with expression (4.24a) leads to the ALE integral form of the mass conservation equation:

$$\left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{V_m(t)} \rho dV + \int_{A_m(t)} \rho \boldsymbol{u}_{conv} \cdot \boldsymbol{n} dA = 0. \tag{4.29}$$

Assuming uniform fields of density ρ and material velocity $\boldsymbol{u}$, it reduces to the *continuous geometric conservation law* (CGL)

$$\left.\frac{\partial}{\partial t}\right|_{\boldsymbol{\chi}} \int_{V_m(t)} dV + \int_{A_m(t)} \boldsymbol{u}_{mesh} \cdot \boldsymbol{n} dA = 0 \tag{4.30}$$

that can also be derived from the ALE integral conservation law for momentum and energy.

The mock-up illustarted in Figure 4.8 has been choosen to check the GCL stated by equation (4.30). In both cases, either with an initialized velocity field or starting from

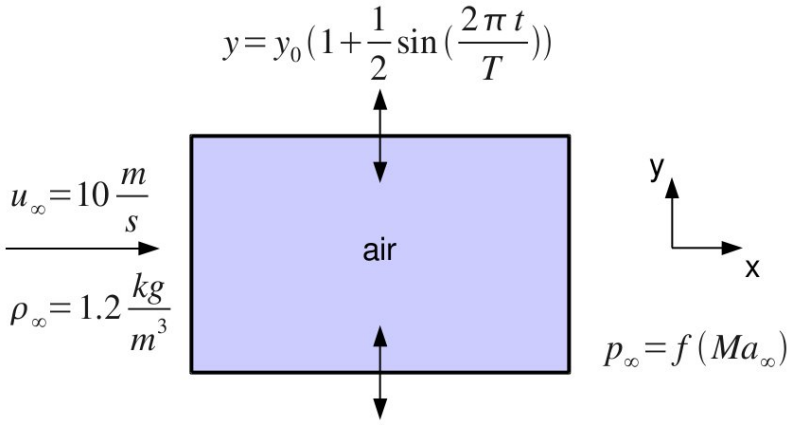


Figure 4.8: Geometric conservation law – tests on a moving grid

zero, the trivial solution of a uniform flow results. Figure 4.9 shows the two extreme positions of the mesh.

Integrating equation (4.30) in time from t^n to t^{n+1} renders the *discrete* geometric conservation law (DCGL)

$$\left|\Omega_{elem}^{n+1}\right| - \left|\Omega_{elem}^{n}\right| = \int_{t^n}^{t^{n+1}} \left(\int_{A_m(t)} \boldsymbol{u}_{mesh} \cdot \boldsymbol{n} dA \right) dt, \tag{4.31}$$

which states that the change in volume (or area in 2D) of each element from t^n to t^{n+1} must be equal to the volume (area respectively) swept by the element boundary during the time interval. Assuming that the volumes Ω_{elem} on the left-hand side of equation (4.31) can be computed exactly, this amounts to requiring the exact computation of the flux on the right-hand side as well. This poses some restrictions on the update procedure for grid position and velocity. Especially in the case of FSI problems, where mesh motion is

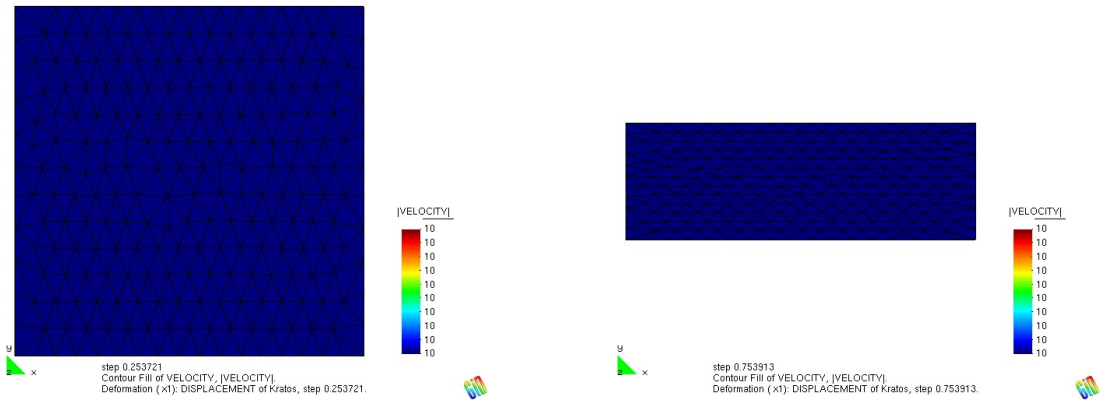


Figure 4.9: GCL – extreme positions of the moving mesh

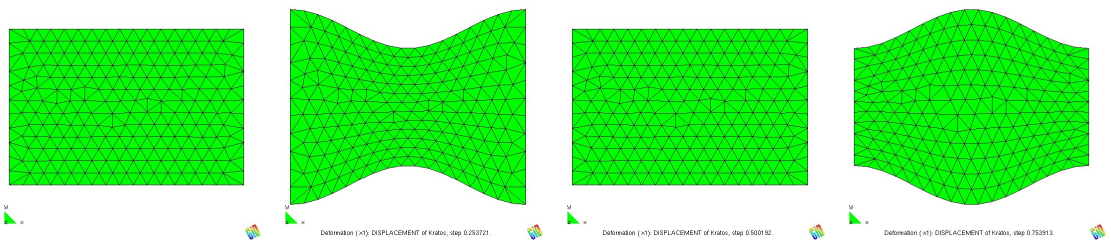
coupled with structural deformation, the intuitive formula for the computation of the mesh velocity (4.27a) is violated in some instances.
It shall be stated here that the practical significance of DGCLs is a debated issue in literature and even in current research the link between DGCLs and the stability (and accuracy) of ALE schemes is still a controversial topic (Donea et al., 2004).

4.3.2. Implementation of Boundary Conditions

For the GCL test case above the implementation of slip respectively no-slip condtions was not affected as the movement of the walls was in a direction orthogonal to the fluid velocity. In the next example we will break up this orthogonality by prescribing the function

$$y(x,t) = \left(1 + \frac{1}{4}\sin\left(\frac{2\pi t}{T}\right) cos\left(\frac{2\pi x}{L}\right)\right) y_0 \tag{4.32}$$

for the nodal positions of the grid points instead. x and y are the nodal coordinates, L is the constant length of the domain and t and T are the simulation time and the duration of one period respectively. Figure 4.10 demonstrates the periodic effect of equation (4.32) on the mesh.



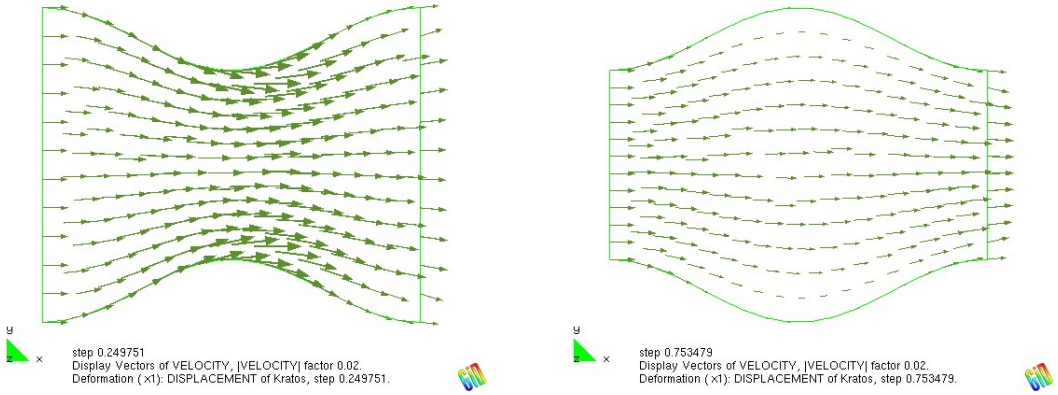
(a) $t = 0,\ t = T$ (b) $t = \frac{1}{4}T$ (c) $t = \frac{1}{2}T$ (d) $t = \frac{3}{4}T$

Figure 4.10: Periodic cycle of mesh motion according to an arbitrary function

This time we have to modify the boundary conditions according to the kinematic requirement (4.25):

$$\boldsymbol{u}_D = \begin{cases} \boldsymbol{u}_{mesh} & \text{for no-slip conditions} \\ (\boldsymbol{u} - \boldsymbol{u}_{mesh}) - (\boldsymbol{u} - \boldsymbol{u}_{mesh}) \cdot \boldsymbol{n} & \text{for slip conditions} \end{cases} \tag{4.33}$$

Figure 4.11 shows the velocity vectors for the two extreme situations of the periodic cycle 4.10 when equation (4.33) is taken into consideration. Whereas the orientation of the velocity vectors follows the boundary movement, their length – indicating the module of the nodal velocities – reflect the equation of continuity, that is to say the conservation of mass.



(a) $t \approx \frac{1}{4}T$ (b) $t \approx \frac{3}{4}T$

Figure 4.11: Slip condition in the case of a moving contour

4.3.3. Interface Variables

So far we only considered the consequences of a moving mesh in terms of nodal displacement (we already mentioned how to calculate `MESH_VELOCITY` from the Kratos variables `DISPLACEMENT`), that is to say the result of a structural deformation on the fluid flow. However, to perform a correct fluid-structure coupling, the reverse has to be taken into account as well. Hence, the force by the fluid flow acting upon the boundary has to be computed and transmitted to the structural application.

In this context, we can re-use a function defined in step 1 of the implemented algorithm: `CalculateRHS` sums up the right-hand side contributions of equation (3.77a) and may be employed, now with the end-of-step values for velocity $\boldsymbol{u}$, density ρ and pressure p, to compute the resulting force on the structure. This one is stored in the Kratos variable `FORCE` so that the structural application can use it in turn.

In addition, the interface has to be marked as such during the pre-processing. This is effectuated by the Kratos variable `IS_INTERFACE` which is delivered to the compressible

fluid application by the respective problemtype. It takes the flag value 1 for fluid-structure interfaces and 0 elsewhere.

4.4. Expectations

Now that the interface variables between structural and flow applications have been defined, everything should be ready for the coupling process. Unfortunately, no coupled FSI simulations could be performed up to now owing to temporal restrictions. Yet we will make up for this in the following weeks and months.

Concerning the *compressible* case, the fluid-structure coupling should be unproblematic and converge within few iterations so that for example aeroelastic simulations on an aircraft wing should work well.
However, in the *incompressible* case we expect some trouble. The balanced mass ratio in water or blood flow simulations will probably lead to situations where the coupling algorithm does not converge.

5 Conclusion

5.1. Résumé of Results

The general algorithm for incompressible and compressible flows proved to deliver satisfactory results for two- and three-dimensional simulations in subsonic regime. From a qualitative point of view the edge-based implementation revealed its advantaged over the element-based one in terms of computational efficiency. A quantitative comparison still has to be done.

Concerning the simulation of FSI problems, the implemented solver has been prepared by modifications due to the ALE formulation of the conservation equations. Some preliminary tests led to positive results so that the coupling within the Kratos environment should work fine.

5.2. Future Prospects

Certainly, performing simulations of real fluid-structure interaction problems is the next important step. A comparison between aeroelastic and hydroelastic phenomena is envisaged, in order to use the flow solver on its whole bandwidth on the one hand and to prove the expected differences between the FSI coupling of compressible and incompressible flows.

With regard to the fluid solver, the consideration of the viscous terms in order to obtain the Navier-Stokes equations seems to be obvious. Nevertheless, a further generalization of the algorithm for perfect gases, so to speak the expansion to the fully compressible regime, appeals far more interesting. This requires the implementation of the energy equation as a further step in the algorithm and some little modifications of the current step 2. In this context shock capturing techniques will be necessary as well to determine the exact shock position in supersonic regime.

A Python Script for ALE Simulation Run

Listing A.1: Start script for a "quasi" FSI simulation

```
1  ##################################################################
2  ## setting the domain size for the problem to be solved
3  domain_size = 2
4
5  ##################################################################
6  ## ATTENTION: here the order is important
7
8  #including kratos path
9  kratos_libs_path = '/usr/local/kratos/kratosR1/libs' ##kratos_root/
       libs
10 #kratos_libs_path = 'C:/kratosR1/libs' ##kratos_root/libs
11 kratos_applications_path = '/usr/local/kratos/kratosR1/applications
       /' ##kratos_root/applications
12 import sys
13 sys.path.append(kratos_libs_path)
14 sys.path.append(kratos_applications_path)
15
16 #importing Kratos main library
17 from Kratos import *
18 kernel = Kernel()   #defining kernel
19
20 #importing applications
21 import applications_interface
22 applications_interface.Import_IncompressibleFluidApplication = True
23 applications_interface.Import_ExternalSolversApplication = False
24 applications_interface.ImportApplications(kernel,
       kratos_applications_path)
25
26 ##################################################################
27 ## from now on the order is not crucial anymore
28
29 from KratosR1IncompressibleFluidApplication import *
```

```
30 ##from KratosR1ExternalSolversApplication import *
31
32 ## defining a model part
33 model_part = ModelPart("FluidPart");
34
35 ## importing the solver files and adding the variables
36 import compressible_fluid_solver
37 compressible_fluid_solver.AddVariables(model_part)
38 model_part.AddNodalSolutionStepVariable(IS_BOUNDARY)
39 model_part.AddNodalSolutionStepVariable(NORMAL)
40 model_part.AddNodalSolutionStepVariable(AUX_INDEX)
41 model_part.AddNodalSolutionStepVariable(EXTERNAL_PRESSURE)
42 model_part.AddNodalSolutionStepVariable(BODY_FORCE)
43 model_part.AddNodalSolutionStepVariable(FRACT_VEL)
44 model_part.AddNodalSolutionStepVariable(MACH_NUMBER)
45 model_part.AddNodalSolutionStepVariable(PRESSURE_COEFFICIENT)
46 model_part.AddNodalSolutionStepVariable(IS_INTERFACE)
47 model_part.AddNodalSolutionStepVariable(DISPLACEMENT)
48 model_part.AddNodalSolutionStepVariable(MESH_VELOCITY)
49 model_part.AddNodalSolutionStepVariable(FORCE)
50
51 ## reading a model
52 gid_io = GidIO("GeomConsLaw",GiDPostMode.GiD_PostBinary)
53 ##gid_io.ReadMesh(model_part.GetMesh())
54 gid_io.ReadModelPart(model_part)
55 gid_io.WriteMesh((model_part).GetMesh(),domain_size,GiDPostMode.
      GiD_PostBinary);
56 print model_part
57
58 ## the buffer size should be set up here after the mesh is read for
       the first time
59 model_part.SetBufferSize(3)
60
61 ## adding degrees of freedom to all of the nodes
62 compressible_fluid_solver.AddDofs(model_part)
63
64 ######################################################################
65 ## settings to be changed
66
67 ## SETTING FREESTREAM CONDITIONS
68 gamma = 1.4
69 Mach = 0.3
70 inlet = Array3()
71 inlet[0] = 10.0
72 inlet[1] = 0.0
73 inlet[2] = 0.0
74 inlet_list = []
75 density = 1.2
76 ext_press = pow(inlet[0]/Mach,2.0) * density/gamma
77 print ext_press
78
```

```
79 ## INITIALIZING FLUID
80 velocity = inlet[0]
81 pressure = ext_press
82 for node in model_part.Nodes:
83     node.SetSolutionStepValue(DENSITY,0,density)
84     node.SetSolutionStepValue(VELOCITY_X,0,velocity)
85     node.SetSolutionStepValue(VELOCITY_Y,0,0.0)
86     node.SetSolutionStepValue(VELOCITY_Z,0,0.0)
87     node.SetSolutionStepValue(PRESSURE,0,pressure)
88
89 ## SETTING BOUNDARY FLAGS
90 #1 - velocity inlet (Dirichlet)
91 #2 - no-slip condition (Dirichlet)
92 #3 - slip condition (Dirichlet)
93 #4 - pressure & slip node
94 #5 - pressure outlet (Neumann)
95
96 ## SETTING BOUNDARY VALUES
97 for node in model_part.Nodes:
98     if(node.GetSolutionStepValue(IS_BOUNDARY) == 1.0):
99         node.SetSolutionStepValue(VELOCITY_X,0,inlet[0])
100        inlet_list.append(node)
101    if(node.GetSolutionStepValue(IS_BOUNDARY) == 4.0 or node.
       GetSolutionStepValue(IS_BOUNDARY) == 5.0):
102        node.SetSolutionStepValue(EXTERNAL_PRESSURE,0,ext_press)
103
104 ## SETTING VELOCITY RAMP-UP
105 ramp_up_steps = 0
106 ramp_up_vel = 10.0
107 initial_dt = 0.001
108
109 ## SETTING SOLVER PARAMETERS
110 CFL_number = 0.5
111 time = 0.0
112 Time = 20.0
113 step = 0
114 tolerance = 1e-3
115 abs_tol = 1e-6
116 n_it_max = 10
117
118 ## SETTING OUTPUT STEPS
119 output_step = 10
120 out = output_step
121
122 ######################################################################
123 ## begin of the simulation run
124
125 matrix_container = MatrixContainer2D()
126 fluid_solver = FluidSolver2D()
127
128 ## finding the neighbours
```

```
129 neighbour_finder = FindNodalNeighboursProcess(model_part,10,10);
130 neighbour_finder.Execute(); ##at wish ... when it is needed
131
132 ## pre-computing edge-data
133 matrix_container.ConstructCSRVector(model_part)
134 matrix_container.BuildCSRData(model_part)
135
136 ## initializing flow solver
137 fluid_solver.Initialize(model_part,matrix_container)
138 fluid_solver.SetFreeFlowConditions(inlet,ext_press,density,gamma)
139 ##fluid_solver.SetAlpha(0.0,model_part.Nodes)
140 pPrecond = DiagonalPreconditioner()
141 ##pPrecond = ILU0Preconditioner()
142 linear_solver = BICGSTABSolver(1e-6,5000,pPrecond)
143 ##linear_solver = CGSolver(1e-6,5000,pPrecond)
144
145 ## time loop
146 while time < Time:
147     ## determining time-step size
148     if(step < ramp_up_steps):
149         delta_t = initial_dt
150         model_part.ProcessInfo[DELTA_TIME] = delta_t
151     else:
152         fluid_solver.ComputeTimeStep(model_part, CFL_number)
153         delta_t = model_part.ProcessInfo[DELTA_TIME]
154
155     ## creating time-step data
156     time = time + delta_t
157     step = step + 1
158     print time
159     model_part.CloneTimeStep(time)
160
161     ## considering velocity ramp-up
162     if(step < ramp_up_steps):
163         inlet[0] = ramp_up_vel * step/ramp_up_steps
164     else:
165         inlet[0] = ramp_up_vel
166     fluid_solver.SetFreeFlowConditions(inlet,ext_press,density,
        gamma)
167
168     ## moving mesh
169     fluid_solver.MoveMesh(model_part)
170     fluid_solver.ComputeMeshVelocity(model_part)
171     matrix_container.BuildCSRData(model_part)
172     fluid_solver.ComputeNormals(model_part)
173
174     ## solving fluid problem
175     if(step > 3):
176         ## STEP 1
177         ratio = 1.0 + tolerance
178         abs_norm = 1.0 + abs_tol
```

```
179         n_it = 0
180         while(ratio > tolerance and abs_norm > abs_tol and n_it <
                n_it_max):
181             norms = fluid_solver.SolveStep1(model_part,
                    matrix_container)
182             if (norms[1] == 0.0):
183                 abs_norm = norms[0]
184             else:
185                 ratio = norms[0]/norms[1]
186                 abs_norm = norms[0]
187             n_it = n_it + 1
188         print "Step 1 cleared"
189         print "    Number of iterations: " + str(n_it)
190         print "    Ratio = " + str(ratio)
191         print "    Absolute difference = " + str(abs_norm)
192 
193         ## STEP 2
194         fluid_solver.SolveStep2(model_part,matrix_container,
                linear_solver)
195         print "Step 2 cleared"
196 
197         ## STEP 3
198         fluid_solver.SolveStep3(model_part,matrix_container)
199         print "Step 3 cleared"
200 
201         ## STEP 4
202         fluid_solver.SolveStep4(model_part.Nodes)
203         print "Step 4 cleared"
204 
205     ## print results
206     if(out == output_step):
207         print "Output"
208         gid_io.WriteNodalResults(AUX_INDEX,model_part.Nodes,time,0)
209         gid_io.WriteNodalResults(IS_BOUNDARY,model_part.Nodes,time
                ,0)
210         gid_io.WriteNodalResults(AUX_INDEX,model_part.Nodes,time,0)
211 
212         gid_io.WriteNodalResults(PRESSURE,model_part.Nodes,time,0)
213         gid_io.WriteNodalResults(VELOCITY,model_part.Nodes,time,0)
214         gid_io.WriteNodalResults(DENSITY,model_part.Nodes,time,0)
215 
216         gid_io.WriteNodalResults(FRACT_VEL,model_part.Nodes,time,0)
217         fluid_solver.CalculateCoefficients(model_part.Nodes)
218         gid_io.WriteNodalResults(PRESSURE_COEFFICIENT,model_part.
                Nodes,time,0)
219         gid_io.WriteNodalResults(MACH_NUMBER,model_part.Nodes,time
                ,0)
220 
221         gid_io.WriteNodalResults(IS_INTERFACE,model_part.Nodes,time
                ,0)
222         gid_io.WriteNodalResults(MESH_VELOCITY,model_part.Nodes,
```

```
            time,0)
223         gid_io.WriteNodalResults(DISPLACEMENT,model_part.Nodes,time
                ,0)
224         gid_io.WriteNodalResults(FORCE,model_part.Nodes,time,0)
225
226         gid_io.Flush()
227         out = 0
228     out = out + 1
229 print "Simulation run terminated correctly"
```

B

C++ Source Code to Compute Edge Data

Listing B.1: Construction of the CSR data vector

```
1  //allocate dynamic memory for the block of CSR data
2  mNonzeroEdgeValues.resize(n_edges);
3  mColumnIndex.resize(n_edges);
4  mRowStartIndex.resize(n_nodes+1);
5
6  //temporary variable as the row start index of a node depends on
       the number of neighbours of the previous one
7  unsigned int row_start_temp = 0;
8
9  //main loop over all nodes
10 for (typename ModelPart::NodesContainerType::iterator node_it=
       model_part.NodesBegin(); node_it!=model_part.NodesEnd(); node_it
       ++)
11 {
12   //get the global index of the node
13   unsigned int i_node = node_it->FastGetSolutionStepValue(AUX_INDEX
         );
14   //determine its neighbours
15   WeakPointerVector< Node<3> >& neighb_nodes = node_it->GetValue(
         NEIGHBOUR_NODES);
16   //number of neighbours determines row start index for the
         following node
17   unsigned int n_neighbours = neighb_nodes.size();
18
19   //reserve memory for work array
20   std::vector<unsigned int> work_array;
21   work_array.reserve(n_neighbours);
22
23   //nested loop over the neighbouring nodes
24   for (WeakPointerVector< Node<3> >::iterator neighb_it=
         neighb_nodes.begin(); neighb_it!=neighb_nodes.end(); neighb_it
         ++)
```

```
25    {
26      //read global index of the neighbouring node
27      work_array.push_back(neighb_it->FastGetSolutionStepValue(
            AUX_INDEX));
28    }
29    //reorder neighbours following their global indices
30    std::sort(work_array.begin(),work_array.end());
31
32    //set current row start index
33    mRowStartIndex[i_node] = row_start_temp;
34    //nested loop over the by now ordered neighbours
35    for (unsigned int counter = 0; counter < n_neighbours; counter++)
36    {
37      //get global index of the neighbouring node
38      unsigned int j_neighbour = work_array[counter];
39      //save column index j of the original matrix
40      mColumnIndex[csr_index] = j_neighbour;
41
42      //calculate CSR index
43      unsigned int csr_index = mRowStartIndex[i_node]+counter;
44      //initialize the CSR vector entries with zero
45      mNonzeroEdgeValues[csr_index].Mass = 0.0;
46      noalias(mNonzeroEdgeValues[csr_index].LaplacianIJ) = ZeroMatrix
            (TDim,TDim);
47      noalias(mNonzeroEdgeValues[csr_index].GradientJ) = ZeroVector(
            TDim);
48      noalias(mNonzeroEdgeValues[csr_index].GradientI) = ZeroVector(
            TDim);
49    }
50    //prepare row start index for next node
51    row_start_temp += n_neighbours;
52  }
53  //add last entry (necessary for abort criterion of the edge loop)
54  mRowStartIndex[n_nodes] = n_edges;
```

Listing B.2: Precompute and assemble edge-based data structure

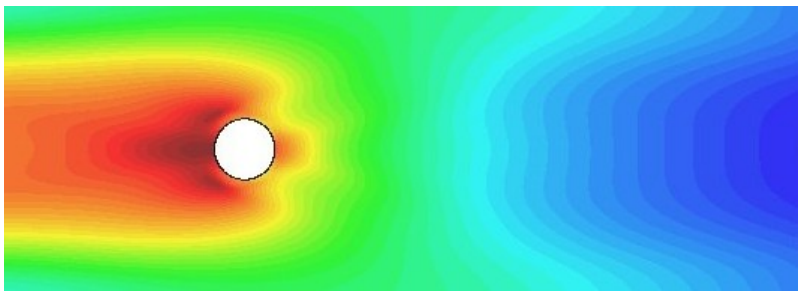
```
//loop over all elements
for (typename ModelPart::ElementsContainerType::iterator elem_it=
    model_part.ElementsBegin(); elem_it!=model_part.ElementsEnd();
    elem_it++)
{
  //get geometry data of the element
  GeometryUtils::CalculateGeometryData(elem_it->GetGeometry(),
     dN_dx, N, volume);

  //set up elemental mass matrices
  CalculateMassMatrix(mass_consistent, volume);
  //compute weighting factor
  //(corresponding to Ni * dOmega respectively Nj * dOmega)
  double weighted_volume = volume / static_cast<double>(TDim+1);

  //loop over the nodes of the element to determine their global
     indices
  for (unsigned int ie_node=0; ie_node<=TDim; ie_node++)
    nodal_indices[ie_node] = elem_it->GetGeometry()[ie_node].
       FastGetSolutionStepValue(AUX_INDEX);

  //assemble global "edge matrices" by adding local contributions
  for (unsigned int ie_node=0; ie_node<=TDim; ie_node++)
    for (unsigned int je_node=0; je_node<=TDim; je_node++)
    {
      //remark: there is no edge linking node i with itself!
      if (ie_node != je_node)
      {
        //calculate CSR index from global index
        unsigned int csr_index = GetCSRIndex(nodal_indices[ie_node
           ], nodal_indices[je_node]);

        //assign precalculated element data to the referring edges
        //contribution to edge mass
        mNonzeroEdgeValues[csr_index].Mass += mass_consistent(
           ie_node,je_node);
        //contribution to edge laplacian
        boost::numeric::ublas::bounded_matrix <double,TDim,TDim>&
           laplacian = mNonzeroEdgeValues[csr_index].LaplacianIJ;
        for (unsigned int k_comp=0; k_comp<TDim; k_comp++)
          for (unsigned int l_comp=0; l_comp<TDim; l_comp++)
            laplacian(k_comp,l_comp) += dN_dx(ie_node,k_comp) *
               dN_dx(je_node,l_comp) * volume;
        //contribution to edge gradient
        array_1d<double, TDim>&  gradient = mNonzeroEdgeValues[
           csr_index].GradientJ;
        for (unsigned int k_comp=0; k_comp<TDim; k_comp++)
          gradient[k_comp] += dN_dx(je_node,k_comp) *
             weighted_volume;
        //contribution to transposed edge gradient
```

```
40          array_1d<double, TDim>& transp_gradient =
                mNonzeroEdgeValues[csr_index].GradientI;
41          for (unsigned int k_comp=0; k_comp<TDim; k_comp++)
42            transp_gradient[k_comp] += dN_dx(ie_node,k_comp) *
                 weighted_volume;
43        }
44      }
45  }
```

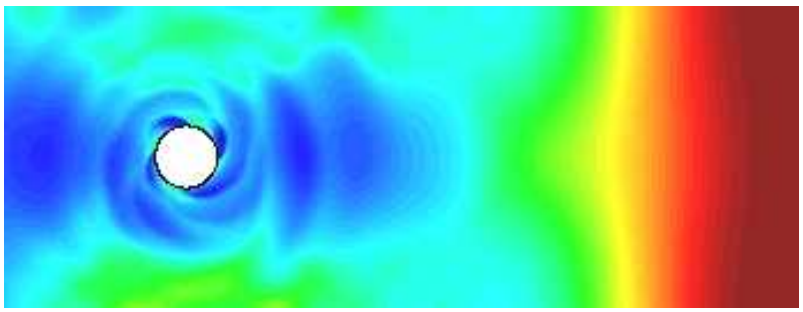
C

Kratos Arts

Jet-Boost



The Swirl



Bibliography

N.A. Adams. Fluidmechanik I – Einführung in die Dynamik der Fluide. Technical report, Lehrstuhl für Aerodynamik, Technische Universität München, 2007.

T. Belytschko, W. K. Liu, and B. Moran. *Nonlinear Finite Elements for Continua and Structures*. John Wiley & Sons, 2000.

S. Candel. Enseignement de Sciences des Transferts: Mécanique des Fluides. Technical report, Laboratoire E.M2.C, Ecole Centrale Paris, 2005.

CIMNE – International Center for Numerical Methods in Engineering. URL http://www.cimne.com.

R. Codina. Pressure stability in fractional step finite element methods for incompressible flows. *Journal of Computational Physics*, volume 170: pp. 112–140, 2001.

R. Codina and A. Folch. A stabilized finite element predictor-corrector scheme for the incompressible Navier-Stokes equations using a nodal-based implementation. *International Journal for Numerical Methods in Fluids*, volume 44: pp. 483–503, 2004.

R. Codina and O. Soto. Approximation of the incompressible Navier-Stokes equations using orthogonal subscale stabilization and pressure segregation on anisotropic finite element meshes. *Computer Methods in Applied Mechanics and Engineering*, volume 193: pp. 1403–1419, 2004.

R. Codina, M. Vázquez, and O.C. Zienkiewicz. A general algorithm for compressible and incompressible flows. Part III: The semi-implicit form. *International Journal for Numerical Methods in Fluids*, volume 27: pp. 13–32, 1998.

P. Dadvand. *A framework for developing finite element codes for multi-disciplinary applications*. PhD thesis, Universitat Politècnica de Catalunya, 2007.

W. Dettmer and D. Perić. A computational framework for fluid-structure interaction: Finite element formulation and applications. *Computer Methods in Applied Mechanics and Engineering*, volume 195: pp. 5754–5779, 2006.

J. Donea and A. Huerta. *Finite Element Methods for Flow Problems*. John Wiley & Sons, 2003.

J. Donea, A. Huerta, J.-Ph. Ponthot, and A. Rodríguez-Ferran. *Encyclopedia of Computational Mechanics*, volume 1: Fundamentals, chapter 14: Arbitrary Lagrangian-Eulerian Methods. John Wiley & Sons, 2004.

GiD – The personal pre- and postprocessor. URL http://www.gidhome.com.

I.J. Keshtiban, F. Belblidia, and M.F. Webster. Compressible flow solvers for low Mach number flows – a review. Technical report, Department of Computer Science, University of Wales, Swansea, 2004.

M. Kohm and J.U. Morawski. KOMA-Script – ein wandelbares LaTeX2ε -Paket, December 2007.

Kratos Trac – Wiki and User Documentation. URL http://kratos.cimne.upc.es/trac/.

R. Löhner. *Applied CFD Techniques - An Introduction based on Finite Element Methods*. John Wiley & Sons, 2001.

R. Löhner. Multistage explicit advective prediction for projection-type incompressible flow solvers. *Journal of Computational Physics*, volume 195: pp. 143–152, 2004.

D.J. Mavriplis and Z. Yang. Construction of the discrete geometric conservation law for high-order time-accurate simulations on dynamic meshes. *Journal of Computational Physics*, volume 213: pp. 557–573, 2006.

D.P. Mok. *Partitionierte Lösungsansätze in der Strukturdynamik und der Fluid-Struktur-Interaktion*. PhD thesis, Universität Stuttgart, 2001.

E. Ortega. *A finite point method for three-dimensional compressible flow*. PhD thesis, Universitat Politècnica de Catalunya, 2007.

E. Ortega, R. Flores, and E. Oñate. An edge-based solver for compressible flows. Technical report, CIMNE, 2005.

R. Rossi. *Light-weight Structures – Numerical Analysis and Coupling Issues*. PhD thesis, Università degli Studi di Padova, 2006.

R. Rossi, S. Idelsohn, and E. Oñate. A new "stabilized" scheme for FSI. 2008.

O. Soto, R. Löhner, J. Cebral, and F. Camelli. A stabilized edge-based implicit incompressible flow formulation. *Computer Methods in Applied Mechanics and Engineering*, volume 193: pp. 2139–2154, 2004.

C++ Language Tutorial, a. URL http://www.cplusplus.com/doc/tutorial/.

Python Tutorial, b. URL http://docs.python.org/tut/.

J. Vierendeels. Implicit Coupling of Partitioned Fluid-Structure Interaction Solvers using Reduced-Order Models. In *Fluid-Structure Interaction: Modelling, Simulation, Optimisation*, volume 53. Springer, 2006.

H. Voß. Math mode. CTAN – the Comprehensive TeX Archive Network, February 2008. Version 2.32.

M. Vázquez, R. Codina, and O.C. Zienkiewicz. *Numerical modelling of compressible laminar and turbulent flow: The CBS algorithm*. CIMNE Monography, 1999.

W.A. Wall. *Fluid-Struktur-Interaktion mit stabilisierten Finiten Elementen*. PhD thesis, Universität Stuttgart, 1999.

J.B. White and P. Sadayappan. On Improving the Performance of Sparse Matrix-Vector Multiplication. In *Proceedings of the Fourth International Conference on High-Performance Computing*, pages pp. 66–71, 1997.

Wikibooks – LaTeX. URL http://en.wikibooks.org/wiki/LaTeX.

Wikipedia – The Free Encyclopedia. URL http://www.wikipedia.org.

O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method*, volume 3: Fluid Dynamics. Butterworth-Heinemann, 5$^{\text{th}}$ edition, 2000.